Advanced Data Structures and Algorithm Analysis

丁尧相
浙江大学

Spring & Summer 2024
Lecture 8
2024-4-15

Outline: Dynamic Programming

- Introduction to DP
- Solving Markov decision process
- Reinforcement learning
- Take-home messages

Outline: Dynamic Programming

- Introduction to DP
- Solving Markov decision process
- Reinforcement learning
- Take-home messages

Dynamic programming history

Bellman. Pioneered the systematic study of dynamic programming in 1950s.

Etymology.

- Dynamic programming = planning over time.
- Secretary of Defense had pathological fear of mathematical research.
- Bellman sought a "dynamic" adjective to avoid conflict.

THE THEORY OF DYNAMIC PROGRAMMING

RICHARD BELLMAN

1. **Introduction.** Before turning to a discussion of some representative problems which will permit us to exhibit various mathematical features of the theory, let us present a brief survey of the fundamental concepts, hopes, and aspirations of dynamic programming.

To begin with, the theory was created to treat the mathematical problems arising from the study of various multi-stage decision processes, which may roughly be described in the following way: We have a physical system whose state at any time t is determined by a set of quantities which we call state parameters, or state variables. At certain times, which may be prescribed in advance, or which may be determined by the process itself, we are called upon to make decisions which will affect the state of the system. These decisions are equivalent to transformations of the state variables, the choice of a decision being identical with the choice of a transformation. The outcome of the preceding decisions is to be used to guide the choice of future ones, with the purpose of the whole process that of maximizing some function of the parameters describing the final state.

Examples of processes fitting this loose description are furnished by virtually every phase of modern life, from the planning of industrial production lines to the scheduling of patients at a medical clinic; from the determination of long-term investment programs for universities to the determination of a replacement policy for machinery in factories; from the programming of training policies for skilled and unskilled labor to the choice of optimal purchasing and inventory policies for department stores and military establishments.

Dynamic programming applications

Application areas.

- Computer science: AI, compilers, systems, graphics, theory, ...
- Operations research.
- Information theory.
- Control theory.
- Bioinformatics.

Some famous dynamic programming algorithms.

- Avidan–Shamir for seam carving.
- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- De Boor for evaluating spline curves.
- Bellman–Ford–Moore for shortest path.
- Knuth–Plass for word wrapping text in TEX.
- Cocke–Kasami–Younger for parsing context-free grammars.
- Needleman–Wunsch/Smith–Waterman for sequence alignment.

Those who cannot remember the past are condemned to repeat it.

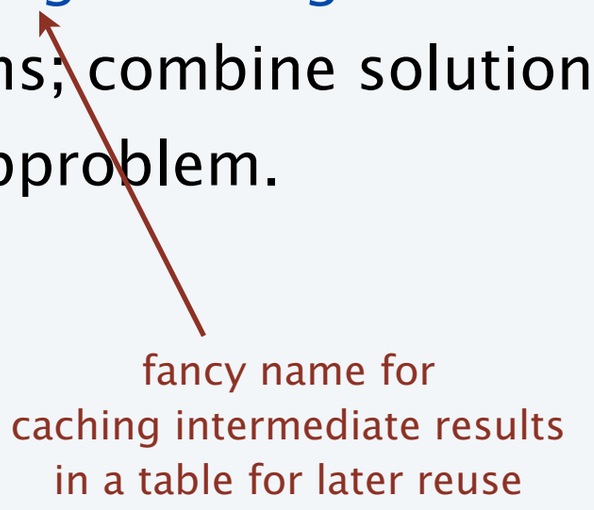
- Dynamic Programming

Algorithmic paradigms

Greed. Process the input in some order, myopically making irrevocable decisions.

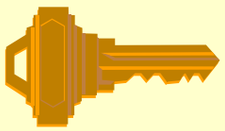
Divide-and-conquer. Break up a problem into independent subproblems; solve each subproblem; combine solutions to subproblems to form solution to original problem.

Dynamic programming. Break up a problem into a series of overlapping subproblems; combine solutions to smaller subproblems to form solution to large subproblem.



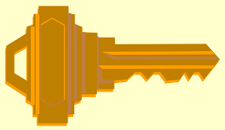
Solve sub-problems just once and save answers in a table

Solve sub-problems just once and save answers in a table



Use a table instead of recursion

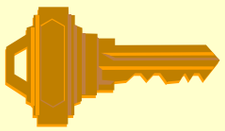
Solve sub-problems just once and save answers in a table



Use a table instead of recursion

1. **Fibonacci Numbers:** $F(N) = F(N-1) + F(N-2)$

Solve sub-problems just once and save answers in a table

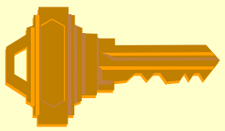


Use a table instead of recursion

1. **Fibonacci Numbers:** $F(N) = F(N-1) + F(N-2)$

```
int  Fib( int N )
{
   if ( N <= 1 )
      return  1;
   else
      return  Fib( N - 1 ) + Fib( N - 2 );
}
```

Solve sub-problems just once and save answers in a table



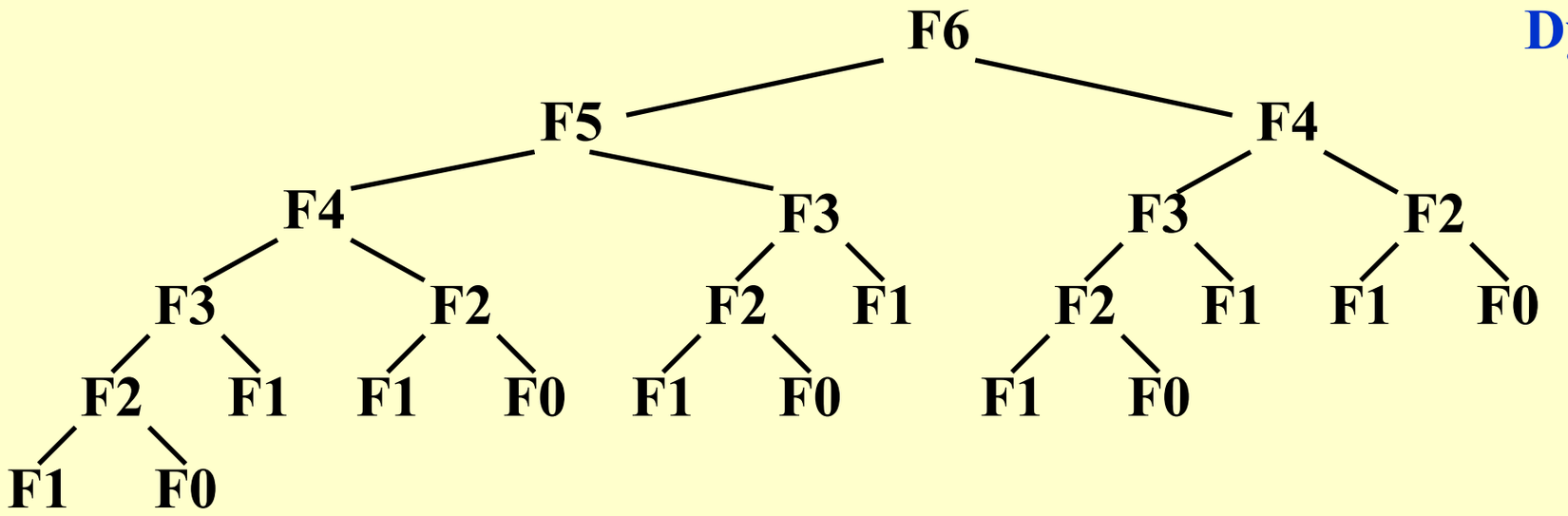
Use a table instead of recursion

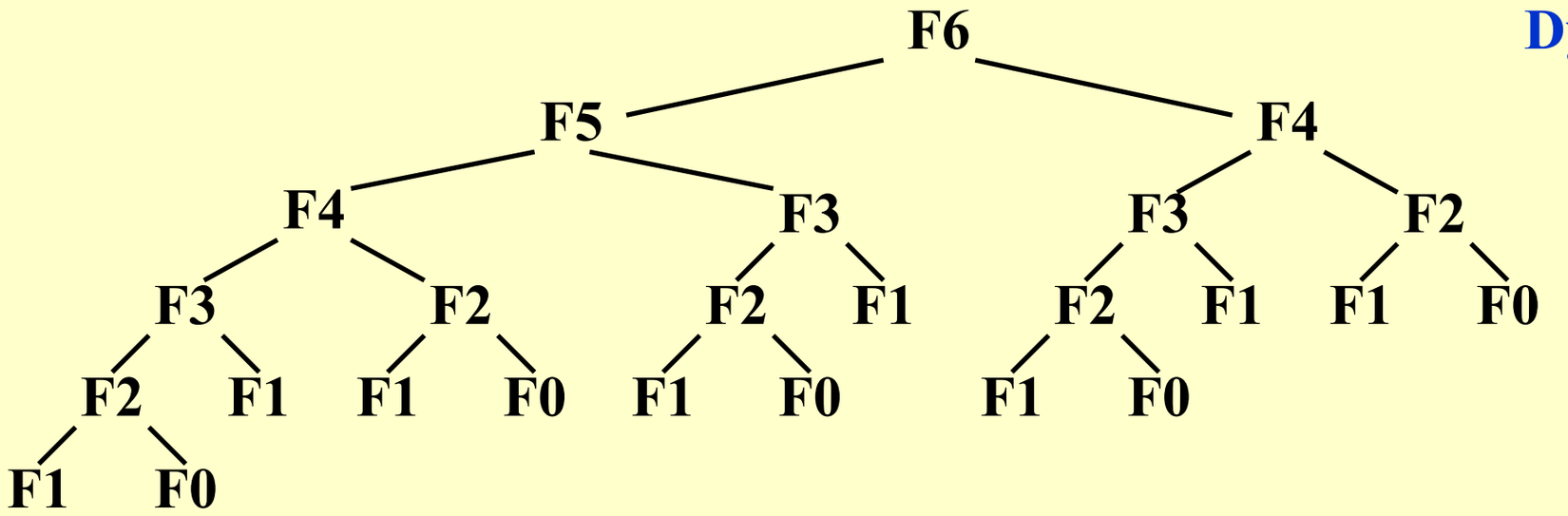
1. **Fibonacci Numbers:** $F(N) = F(N-1) + F(N-2)$

```
int  Fib( int N )
{
    if ( N <= 1 )
        return  1;
    else
        return  Fib( N - 1 ) + Fib( N - 2 );
}
```

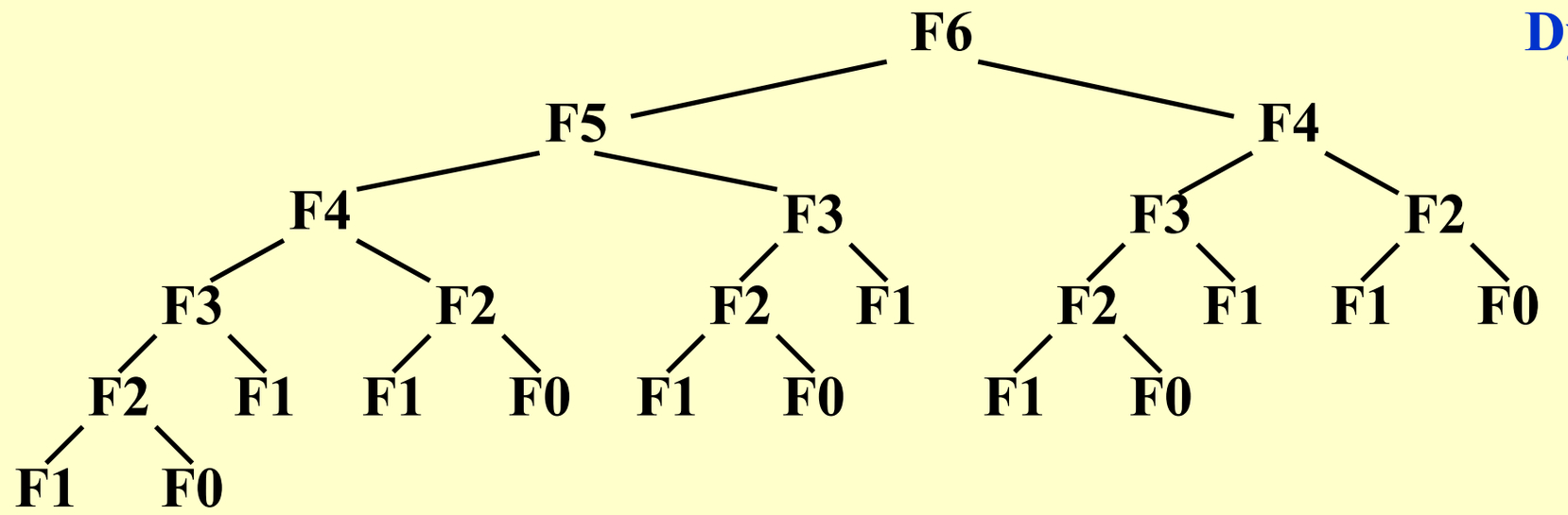
$$T(N) \geq T(N-1) + T(N-2)$$

$$\rightarrow \quad T(N) \geq F(N)$$


F6
F5
F4
F4
F3
F3
F2
F3
F2
F2
F1
F2
F1
F1
F0
F2
F1
F1
F0
F1
F0
F1
F0
F1
F0

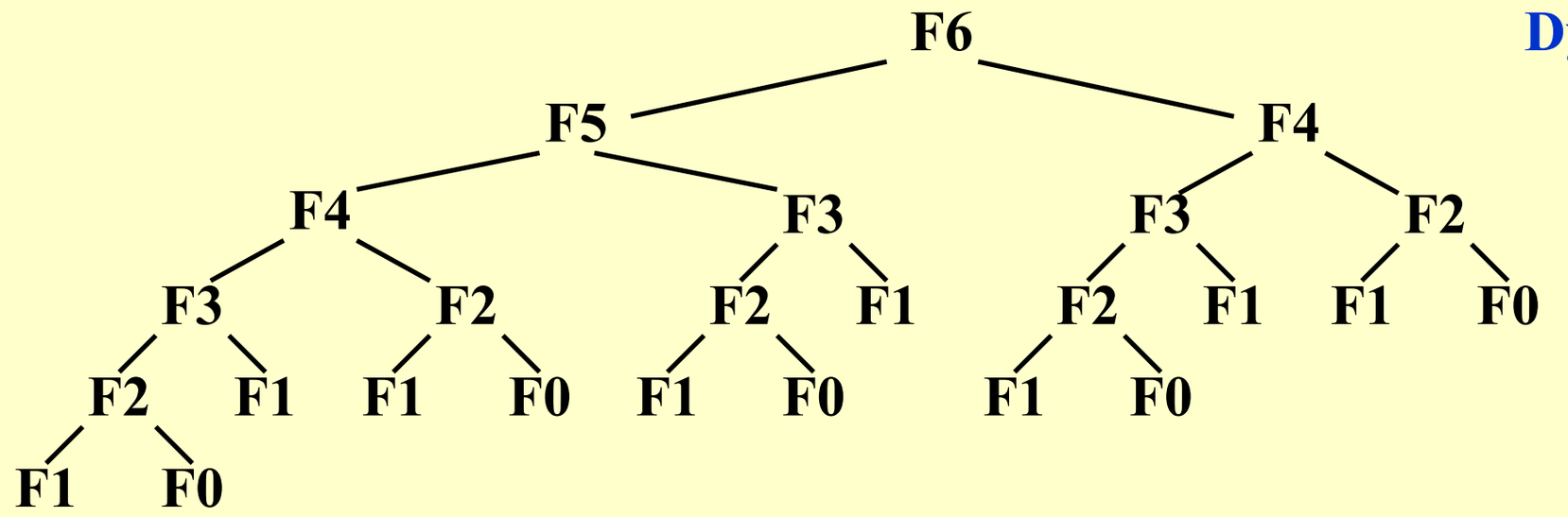


Trouble-maker: The growth of redundant calculations is explosive.



Trouble-maker: **The growth of redundant calculations is explosive.**

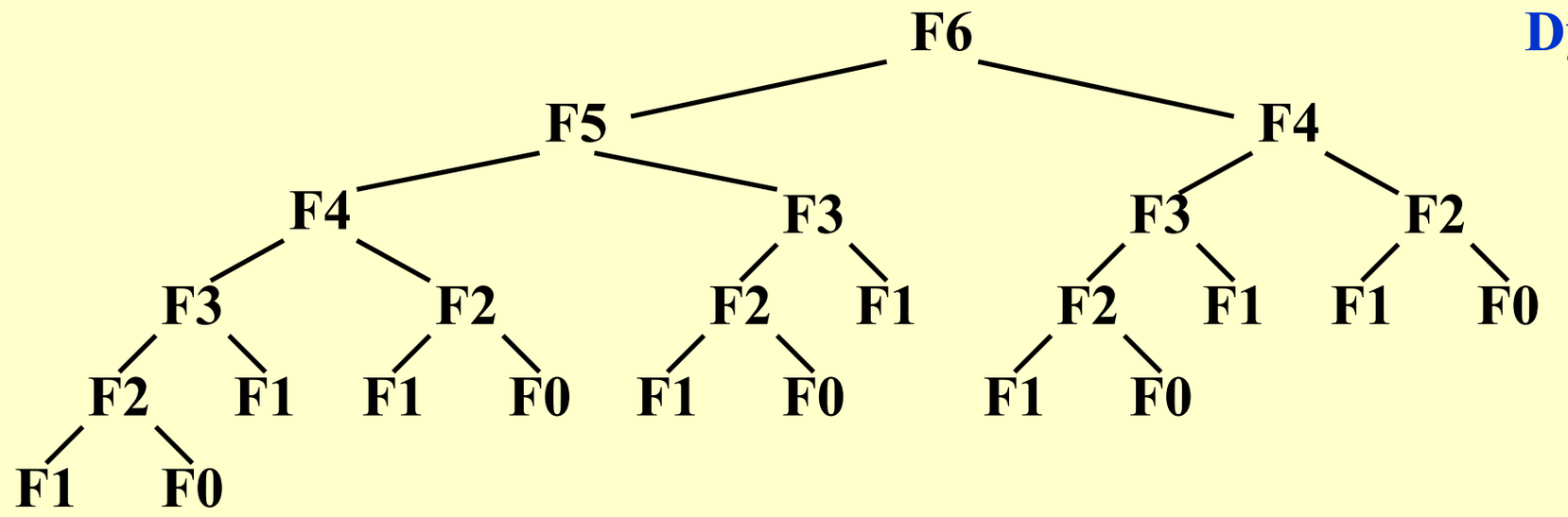
Solution: **Record the two most recently computed values to avoid recursive calls.**



Trouble-maker: **The growth of redundant calculations is explosive.**

Solution: **Record the two most recently computed values to avoid recursive calls.**

```
int  Fibonacci ( int N )
{   int  i, Last, NextToLast, Answer;
    if ( N <= 1 )  return  1;
    Last = NextToLast = 1;    /* F(0) = F(1) = 1 */
    for ( i = 2; i <= N; i++ ) {
        Answer = Last + NextToLast;   /* F(i) = F(i-1) + F(i-2) */
        NextToLast = Last; Last = Answer;  /* update F(i-1) and F(i-2) */
    }  /* end-for */
    return  Answer;
}
```



Trouble-maker: **The growth of redundant calculations is explosive.**

Solution: **Record the two most recently computed values to avoid recursive calls.**

```
int  Fibonacci ( int N )
{   int  i, Last, NextToLast, Answer;
    if ( N <= 1 )  return  1;
    Last = NextToLast = 1;    /* F(0) = F(1) = 1 */
    for ( i = 2; i <= N; i++ ) {
        Answer = Last + NextToLast;   /* F(i) = F(i-1) + F(i-2) */
        NextToLast = Last; Last = Answer;  /* update F(i-1) and F(i-2) */
    }  /* end-for */
    return  Answer;
}
```

$$T(N) = \mathrm{O}(N)$$

2. Ordering Matrix Multiplications

2. Ordering Matrix Multiplications

〖Example〗 Suppose we are to multiply 4 matrices

$M_{1\,[\,10\times20\,]} * M_{2\,[\,20\times50\,]} * M_{3\,[\,50\times1\,]} * M_{4\,[\,1\times100\,]}$.

2. Ordering Matrix Multiplications

〖Example〗 Suppose we are to multiply 4 matrices

$$M_{1\,[\,10\times20\,]} * M_{2\,[\,20\times50\,]} * M_{3\,[\,50\times1\,]} * M_{4\,[\,1\times100\,]}\,.$$

If we multiply in the order

$$M_{1\,[\,10\times20\,]} * (M_{2\,[\,20\times50\,]} * (M_{3\,[\,50\times1\,]} * M_{4\,[\,1\times100\,]}))$$

Then the computing time is

$$50 \times 1 \times 100 + 20 \times 50 \times 100 + 10 \times 20 \times 100 = 125{,}000$$

2. Ordering Matrix Multiplications

〖Example〗 Suppose we are to multiply 4 matrices

$$M_{1\,[\,10\times20\,]} * M_{2\,[\,20\times50\,]} * M_{3\,[\,50\times1\,]} * M_{4\,[\,1\times100\,]} \cdot$$

If we multiply in the order

$$M_{1\,[\,10\times20\,]} * (M_{2\,[\,20\times50\,]} * (M_{3\,[\,50\times1\,]} * M_{4\,[\,1\times100\,]}))$$

Then the computing time is

$$50 \times 1 \times 100 + 20 \times 50 \times 100 + 10 \times 20 \times 100 = 125{,}000$$

If we multiply in the order

$$(M_{1\,[\,10\times20\,]} * (M_{2\,[\,20\times50\,]} * M_{3\,[\,50\times1\,]})) * M_{4\,[\,1\times100\,]}$$

Then the computing time is

$$20 \times 50 \times 1 + 10 \times 20 \times 1 + 10 \times 1 \times 100 = 2{,}200$$

2. Ordering Matrix Multiplications

〖Example〗 Suppose we are to multiply 4 matrices

$$M_{1\,[\,10\times20\,]} * M_{2\,[\,20\times50\,]} * M_{3\,[\,50\times1\,]} * M_{4\,[\,1\times100\,]}\,.$$

If we multiply in the order

$$M_{1\,[\,10\times20\,]} * (M_{2\,[\,20\times50\,]} * (M_{3\,[\,50\times1\,]} * M_{4\,[\,1\times100\,]}))$$

Then the computing time is

$$50 \times 1 \times 100 + 20 \times 50 \times 100 + 10 \times 20 \times 100 = 125{,}000$$

If we multiply in the order

$$(M_{1\,[\,10\times20\,]} * (M_{2\,[\,20\times50\,]} * M_{3\,[\,50\times1\,]})) * M_{4\,[\,1\times100\,]}$$

Then the computing time is

$$20 \times 50 \times 1 + 10 \times 20 \times 1 + 10 \times 1 \times 100 = 2{,}200$$

Problem: In which order can we compute the product of n matrices with minimal computing time?

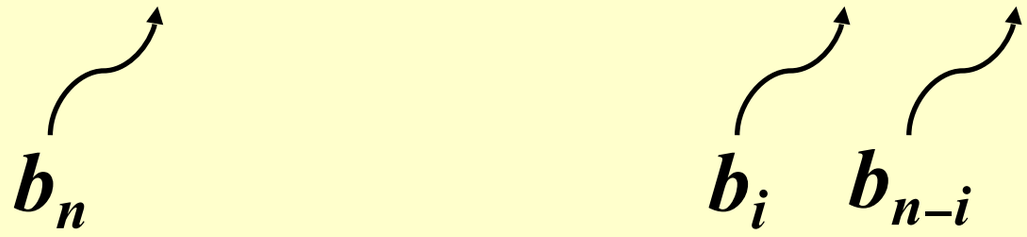
Let b_n = number of different ways to compute $M_1 \cdot M_2 \cdot \cdots \cdot M_n$. Then we have $b_2 = 1, b_3 = 2, b_4 = 5, \cdots$

Let b_n = number of different ways to compute $M_1 \cdot M_2 \cdot \cdots \cdot M_n$. Then we have $b_2 = 1, b_3 = 2, b_4 = 5, \cdots$

Let $M_{ij} = M_i \cdot \cdots \cdot M_j$. Then $M_{1n} = M_1 \cdot \cdots \cdot M_n = M_{1i} \cdot M_{i+1\ n}$

Let b_n = number of different ways to compute $M_1 \cdot M_2 \cdot \cdots \cdot M_n$. Then we have $b_2 = 1, b_3 = 2, b_4 = 5, \cdots$

Let $M_{ij} = M_i \cdot \cdots \cdot M_j$. Then $M_{1n} = M_1 \cdot \cdots \cdot M_n = M_{1i} \cdot M_{i+1\ n}$



Let b_n = number of different ways to compute $M_1 \cdot M_2 \cdot \cdots \cdot M_n$. Then we have $b_2 = 1, b_3 = 2, b_4 = 5, \cdots$

Let $M_{ij} = M_i \cdot \cdots \cdot M_j$. Then $M_{1n} = M_1 \cdot \cdots \cdot M_n = M_{1i} \cdot M_{i+1\ n}$

$$\Rightarrow \quad b_n = \sum_{i=1}^{n-1} b_i b_{n-i} \quad \text{where } n > 1 \text{ and } b_1 = 1.$$

Let b_n = number of different ways to compute $M_1 \cdot M_2 \cdot \cdots \cdot M_n$. Then we have $b_2 = 1, b_3 = 2, b_4 = 5, \cdots$

Let $M_{ij} = M_i \cdot \cdots \cdot M_j$. Then $M_{1n} = M_1 \cdot \cdots \cdot M_n = M_{1i} \cdot M_{i+1\ n}$

$$\Rightarrow \quad b_n = \sum_{i=1}^{n-1} b_i b_{n-i} \quad \text{where } n > 1 \text{ and } b_1 = 1.$$

$$b_n = O\left(\frac{4^n}{n\sqrt{n}}\right) \quad \text{/* Catalan number */}$$

Let b_n = number of different ways to compute $M_1 \cdot M_2 \cdot \cdots \cdot M_n$. Then we have $b_2 = 1, b_3 = 2, b_4 = 5, \cdots$

Let $M_{ij} = M_i \cdot \cdots \cdot M_j$. Then $M_{1n} = M_1 \cdot \cdots \cdot M_n = M_{1i} \cdot M_{i+1\ n}$

$$\Rightarrow \quad b_n = \sum_{i=1}^{n-1} b_i b_{n-i} \quad \text{where } n > 1 \text{ and } b_1 = 1.$$

$$b_n = O\left(\frac{4^n}{n\sqrt{n}}\right) \quad \text{/* Catalan number */}$$

Suppose we are to multiply n matrices $M_1 * \cdots \cdots * M_n$ where M_i is an $r_{i-1} \times r_i$ matrix. Let m_{ij} be the cost of the optimal way to compute $M_i * \cdots \cdots * M_j$. Then we have the recurrence equations:

Let b_n = number of different ways to compute $M_1 \cdot M_2 \cdot \cdots \cdot M_n$. Then we have $b_2 = 1, b_3 = 2, b_4 = 5, \cdots$

Let $M_{ij} = M_i \cdot \cdots \cdot M_j$. Then $M_{1n} = M_1 \cdot \cdots \cdot M_n = M_{1i} \cdot M_{i+1\ n}$

$$\Rightarrow \quad b_n = \sum_{i=1}^{n-1} b_i b_{n-i} \quad \text{where } n > 1 \text{ and } b_1 = 1.$$

$$b_n = O\left(\frac{4^n}{n\sqrt{n}}\right) \quad \text{/* Catalan number */}$$

Suppose we are to multiply n matrices $M_1 * \cdots \cdots * M_n$ where M_i is an $r_{i-1} \times r_i$ matrix. Let m_{ij} be the cost of the optimal way to compute $M_i * \cdots \cdots * M_j$. Then we have the recurrence equations:

$$m_{ij} = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le l < j} \{ m_{il} + m_{l+1\ j} + r_{i-1} r_l r_j \} & \text{if } j > i \end{cases}$$

Let b_n = number of different ways to compute $M_1 \cdot M_2 \cdot \cdots \cdot M_n$. Then we have $b_2 = 1, b_3 = 2, b_4 = 5, \cdots$

Let $M_{ij} = M_i \cdot \cdots \cdot M_j$. Then $M_{1n} = M_1 \cdot \cdots \cdot M_n = M_{1i} \cdot M_{i+1\ n}$

$$\Rightarrow \quad b_n = \sum_{i=1}^{n-1} b_i b_{n-i} \quad \text{wh}$$

$$b_n = O\left(\frac{4^n}{n\sqrt{n}}\right)$$

There are only O(N^2) values of M_{ij} .

Suppose we are to multiply n matrices $M_1 * \cdots \cdots * M_n$ where M_i is an $r_{i-1} \times r_i$ matrix. Let m_{ij} be the cost of the optimal way to compute $M_i * \cdots \cdots * M_j$. Then we have the recurrence equations:

$$m_{ij} = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le l < j} \{ m_{il} + m_{l+1\ j} + r_{i-1} r_l r_j \} & \text{if } j > i \end{cases}$$

Let b_n = number of different ways to compute $M_1 \cdot M_2 \cdot \cdots \cdot M_n$. Then we have $b_2 = 1, b_3 = 2, b_4 = 5, \cdots$

Let $M_{ij} = M_i \cdot \cdots \cdot M_j$. Then $M_{1n} = M_1 \cdot \cdots \cdot M_n = M_{1i} \cdot M_{i+1\ n}$

$$\Rightarrow \quad b_n = \sum_{i=1}^{n-1} b_i b_{n-i} \quad \text{wh}$$

$$b_n = O\left(\frac{4^n}{n\sqrt{n}}\right)$$

If $j - i = k$, then the only values M_{xy} required to compute M_{ij} satisfy $y - x < k$.

Suppose we are to multiply n matrices $M_1 * \cdots \cdots * M_n$ where M_i is an $r_{i-1} \times r_i$ matrix. Let m_{ij} be the cost of the optimal way to compute $M_i * \cdots \cdots * M_j$. Then we have the recurrence equations:

$$m_{ij} = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le l < j} \{ m_{il} + m_{l+1\ j} + r_{i-1} r_l r_j \} & \text{if } j > i \end{cases}$$

```
/* r contains number of columns for each of the N matrices */
/* r[ 0 ] is the number of rows in matrix 1 */
/* Minimum number of multiplications is left in M[ 1 ][ N ] */
void OptMatrix( const long r[ ], int N, TwoDimArray M )
{   int  i, j, k, L;
    long  ThisM;
    for( i = 1; i <= N; i++ )   M[ i ][ i ] = 0;
    for( k = 1; k < N; k++ ) /* k = j - i */
        for( i = 1; i <= N - k; i++ ) { /* For each position */
            j = i + k;    M[ i ][ j ] = Infinity;
            for( L = i; L < j; L++ ) {
                ThisM = M[ i ][ L ] + M[ L + 1 ][ j ]
                        + r[ i - 1 ] * r[ L ] * r[ j ];
                if ( ThisM < M[ i ][ j ] )  /* Update min */
                        M[ i ][ j ] = ThisM;
            } /* end for-L */
        } /* end for-Left */
}
```

```
/* r contains number of columns for each of the N matrices */
/* r[ 0 ] is the number of rows in matrix 1 */
/* Minimum number of multiplications is left in M[ 1 ][ N ] */
void OptMatrix( const long r[ ], int N, TwoDimArray M )
{   int  i, j, k, L;
    long  ThisM;
    for( i = 1; i <= N; i++ )   M[ i ][ i ] = 0;
    for( k = 1; k < N; k++ ) /* k = j - i */
        for( i = 1; i <= N - k; i++ ) { /* For each position */
            j = i + k;    M[ i ][ j ] = Infinity;
            for( L = i; L < j; L++ ) {
                ThisM = M[ i ][ L ] + M[ L + 1 ][ j ]
                        + r[ i - 1 ] * r[ L ] * r[ j ];
                if ( ThisM < M[ i ][ j ] )  /* Update min */
                    M[ i ][ j ] = ThisM;
            } /* end for-L */
        } /* end for-Left */
}
```

$$m_{ij} = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le l < j} \{ m_{il} + m_{l+1\,j} + r_{i-1} r_l r_j \} & \text{if } j > i \end{cases}$$

```
/* r contains number of columns for each of the N matrices */
/* r[ 0 ] is the number of rows in matrix 1 */
/* Minimum number of multiplications is left in M[ 1 ][ N ] */
void OptMatrix( const long r[ ], int N, TwoDimArray M )
{   int  i, j, k, L;
    long  ThisM;
    for( i = 1; i <= N; i++ )   M[ i ][ i ] = 0;
    for( k = 1; k < N; k++ ) /* k = j - i */
        for( i = 1; i <= N - k; i++ ) { /* For each position */
            j = i + k;    M[ i ][ j ] = Infinity;
            for( L = i; L < j; L++ ) {
                ThisM = M[ i ][ L ] + M[ L + 1 ][ j ]
                        + r[ i - 1 ] * r[ L ] * r[ j ];
                if ( ThisM < M[ i ][ j ] )  /* Update min */
                    M[ i ][ j ] = ThisM;
            } /* end for-L */
        } /* end for-Left */
}
```

$$\begin{matrix} m_{1,N} & & & & \\ m_{1,N-1} & m_{2,N} & & & \\ \vdots & \vdots & \ddots & & \\ m_{1,2} & m_{2,3} & \cdots & m_{N-1,N} & \\ m_{1,1} & m_{2,2} & \cdots & m_{N-1,N-1} & m_{N,N} \end{matrix}$$

$$m_{ij} = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le l < j} \{ m_{il} + m_{l+1\,j} + r_{i-1} r_l r_j \} & \text{if } j > i \end{cases}$$

```
/* r contains number of columns for each of the N matrices */
/* r[ 0 ] is the number of rows in matrix 1 */
/* Minimum number of multiplications is left in M[ 1 ][ N ] */
void OptMatrix( const long r[ ], int N, TwoDimArray M )
{   int  i, j, k, L;
    long  ThisM;
    for( i = 1; i <= N; i++ )   M[ i ][ i ] = 0;
    for( k = 1; k < N; k++ ) /* k = j - i */
        for( i = 1; i <= N - k; i++ ) { /* For each position */
            j = i + k;    M[ i ][ j ] = Infinity;
            for( L = i; L < j; L++ ) {
                ThisM = M[ i ][ L ] + M[ L + 1 ][ j ]
                        + r[ i - 1 ] * r[ L ] * r[ j ];
                if ( ThisM < M[ i ][ j ] )  /* Update min */
                    M[ i ][ j ] = ThisM;
            } /* end for-L */
        } /* end for-Left */
}
```

$$\begin{matrix} m_{1,N} & & & & \\ m_{1,N-1} & m_{2,N} & & & \\ \vdots & \vdots & \ddots & & \\ m_{1,2} & m_{2,3} & \cdots & m_{N-1,N} & \\ m_{1,1} & m_{2,2} & \cdots & m_{N-1,N-1} & m_{N,N} \end{matrix}$$

$$T(N) = \mathrm{O}(N^3)$$

$$m_{ij} = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le l < j} \{ m_{il} + m_{l+1\,j} + r_{i-1} r_l r_j \} & \text{if } j > i \end{cases}$$

```
/* r contains number of columns for each of the N matrices */
/* r[ 0 ] is the number of rows in matrix 1 */
/* Minimum number of multiplications is left in M[ 1 ][ N ] */
void OptMatrix( const long r[ ], int N, TwoDimArray M )
{   int  i, j, k, L;
    long  ThisM;
    for( i = 1; i <= N; i++ )   M[ i ][ i ] = 0;
    for( k = 1; k < N; k++ ) /* k = j - i */
        for( i = 1; i <= N - k; i++ ) { /* For each position */
            j = i + k;    M[ i ][ j ] = Infinity;
            for( L = i; L < j; L++ ) {
                ThisM = M[ i ][ L ] + M[ L + 1 ][ j ]
                        + r[ i - 1 ] * r[ L ] * r[ j ];
                if ( ThisM < M[ i ][ j ] )  /* Update min */
                    M[ i ][ j ] = ThisM;
            } /* end for-L */
        } /* end for-Left */
}
```

$$\begin{array}{ccccc} m_{1,N} & & & & \\ m_{1,N-1} & m_{2,N} & & & \\ \vdots & \vdots & \ddots & & \\ m_{1,2} & m_{2,3} & \cdots & m_{N-1,N} & \\ m_{1,1} & m_{2,2} & \cdots & m_{N-1,N-1} & m_{N,N} \end{array}$$

$$T(N) = \mathrm{O}(N^3)$$

To record the ordering please refer to Weiss Figure 10.46 on p.388

```
MATRIX-CHAIN-ORDER(p, n)
 1  let m[1:n, 1:n] and s[1:n − 1, 2:n] be new tables
 2  for i = 1 to n                          // chain length 1
 3      m[i, i] = 0
 4  for l = 2 to n                          // l is the chain length
 5      for i = 1 to n − l + 1              // chain begins at A_i
 6          j = i + l − 1                   // chain ends at A_j
 7          m[i, j] = ∞
 8          for k = i to j − 1              // try A_{i:k} A_{k+1:j}
 9              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10              if q < m[i, j]
11                  m[i, j] = q             // remember this cost
12                  s[i, j] = k             // remember this index
13  return m and s
```

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2      print "A"_i
3  else print "("
4      PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5      PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6      print ")"
```

```
MATRIX-CHAIN-ORDER(p, n)
 1  let m[1:n, 1:n] and s[1:n − 1, 2:n] be new tables
 2  for i = 1 to n                          // chain length 1
 3      m[i, i] = 0
 4  for l = 2 to n                          // l is the chain length
 5      for i = 1 to n − l + 1              // chain begins at A_i
 6          j = i + l − 1                   // chain ends at A_j
 7          m[i, j] = ∞
 8          for k = i to j − 1              // try A_{i:k} A_{k+1:j}
 9              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10              if q < m[i, j]
11                  m[i, j] = q             // remember this cost
12                  s[i, j] = k             // remember this index
13  return m and s
```

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2      print "A"_i
3  else print "("
4      PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5      PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6      print ")"
```

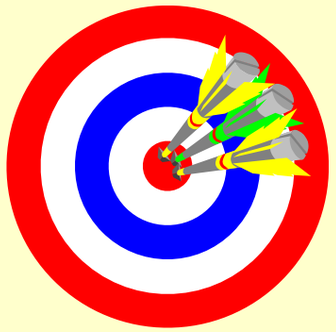
CLRS (4th edition) pp. 378, 381

3. Optimal Binary Search Tree

—— The best for static searching (without insertion and deletion)

3. Optimal Binary Search Tree

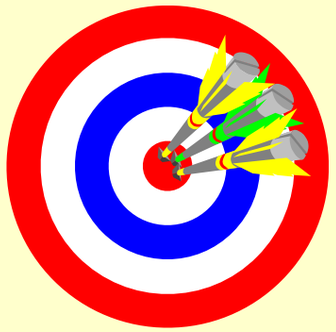
—— The best for static searching (without insertion and deletion)



Given N words $w_1 < w_2 < \ldots\ldots < w_N$, and the probability of searching for each w_i is p_i. Arrange these words in a binary search tree in a way that minimize the expected total access time.

3. Optimal Binary Search Tree

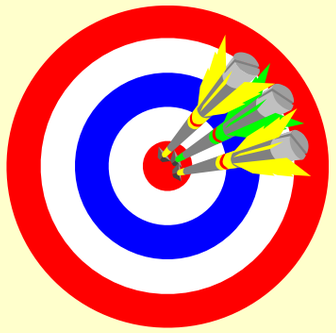
—— The best for static searching (without insertion and deletion)



Given N words $w_1 < w_2 < \ldots\ldots < w_N$, and the probability of searching for each w_i is p_i. Arrange these words in a binary search tree in a way that minimize the expected total access time. $T(N) = \sum_{i=1}^{N} p_i \cdot (1 + d_i)$

3. Optimal Binary Search Tree

—— The best for static searching (without insertion and deletion)



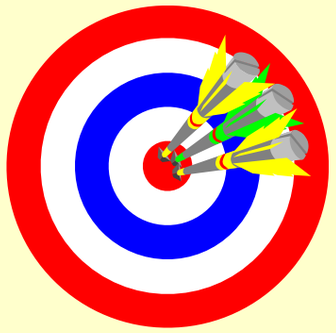
Given N words $w_1 < w_2 < \ldots\ldots < w_N$, and the probability of searching for each w_i is p_i. Arrange these words in a binary search tree in a way that minimize the expected total access time. $T(N) = \sum_{i=1}^{N} p_i \cdot (1 + d_i)$

〖Example〗 Given the following table of probabilities:

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

3. Optimal Binary Search Tree

—— The best for static searching (without insertion and deletion)



Given N words $w_1 < w_2 < \ldots\ldots < w_N$, and the probability of searching for each w_i is p_i. Arrange these words in a binary search tree in a way that minimize the expected total access time. $T(N) = \sum_{i=1}^{N} p_i \cdot (1 + d_i)$

〖Example〗 Given the following table of probabilities:

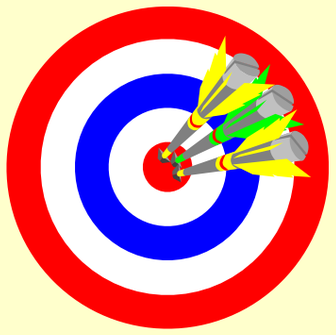
word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

Discussion 10:

Please draw the trees obtained by greedy methods and by AVL rotations. What are their expected total access times?

3. Optimal Binary Search Tree

—— The best for static searching (without insertion and deletion)



Given N words $w_1 < w_2 < \ldots\ldots < w_N$, and the probability of searching for each w_i is p_i. Arrange these words in a binary search tree in a way that minimize the expected total access time. $T(N) = \sum_{i=1}^{N} p_i \cdot (1 + d_i)$

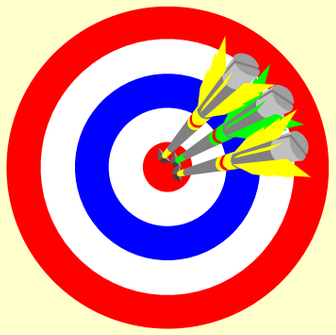
〖Example〗 Given the following table of probabilities:

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

Greedy Method

3. Optimal Binary Search Tree

—— The best for static searching (without insertion and deletion)

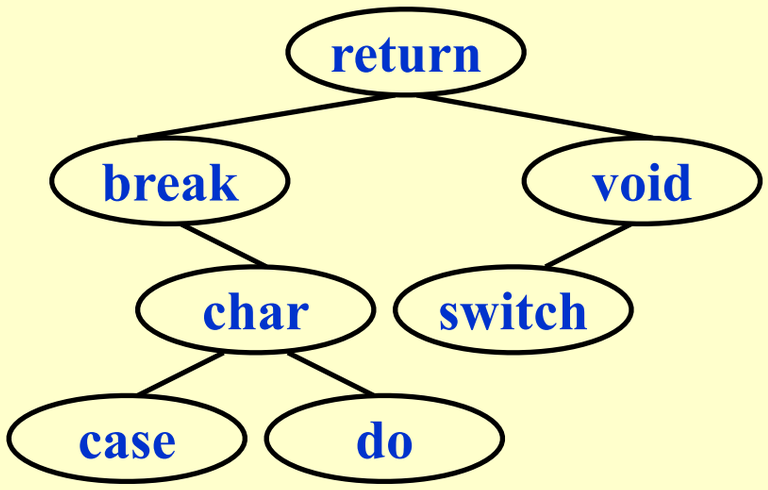


Given N words $w_1 < w_2 < \ldots\ldots < w_N$, and the probability of searching for each w_i is p_i. Arrange these words in a binary search tree in a way that minimize the expected total access time. $T(N) = \sum_{i=1}^{N} p_i \cdot (1 + d_i)$

〖Example〗 Given the following table of probabilities:

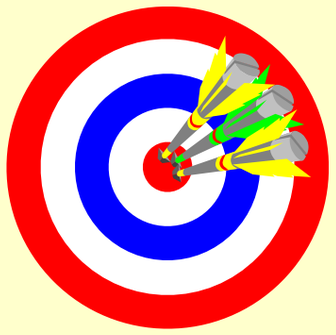
word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

Greedy Method



3. Optimal Binary Search Tree

—— The best for static searching (without insertion and deletion)

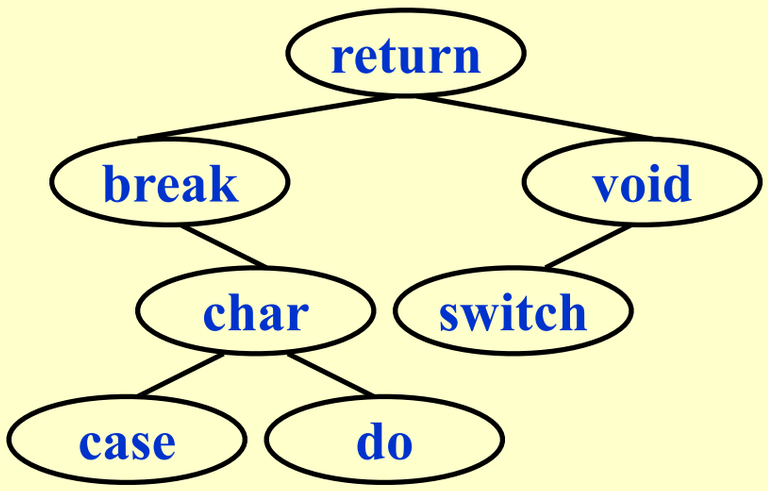


Given N words $w_1 < w_2 < \ldots\ldots < w_N$, and the probability of searching for each w_i is p_i. Arrange these words in a binary search tree in a way that minimize the expected total access time. $T(N) = \sum_{i=1}^{N} p_i \cdot (1 + d_i)$

〖Example〗 Given the following table of probabilities:

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

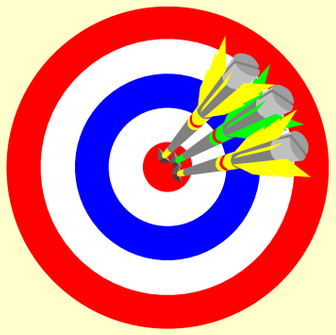
Greedy Method



Time = 2.43

3. Optimal Binary Search Tree

—— The best for static searching (without insertion and deletion)

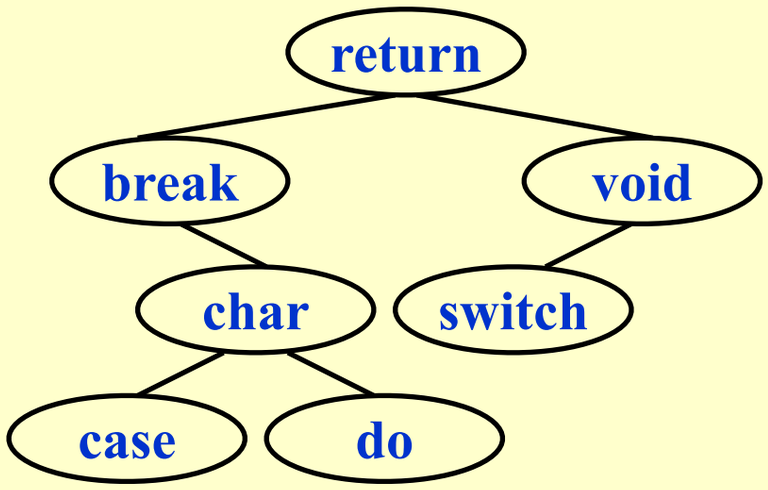


Given N words $w_1 < w_2 < \ldots\ldots < w_N$, and the probability of searching for each w_i is p_i. Arrange these words in a binary search tree in a way that minimize the expected total access time. $T(N) = \sum_{i=1}^{N} p_i \cdot (1 + d_i)$

〖Example〗 Given the following table of probabilities:

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

Greedy Method

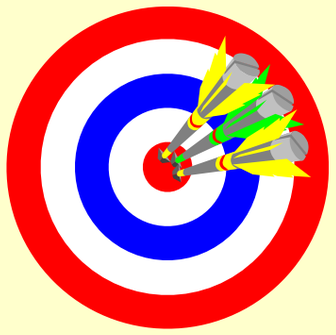


Time = 2.43

AVL Method

3. Optimal Binary Search Tree

—— The best for static searching (without insertion and deletion)

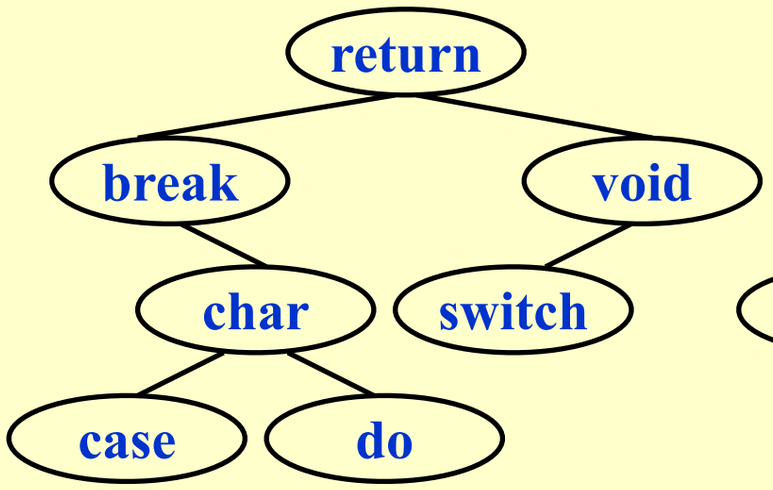


Given N words $w_1 < w_2 < \ldots\ldots < w_N$, and the probability of searching for each w_i is p_i. Arrange these words in a binary search tree in a way that minimize the expected total access time. $T(N) = \sum_{i=1}^{N} p_i \cdot (1 + d_i)$

〖Example〗 Given the following table of probabilities:

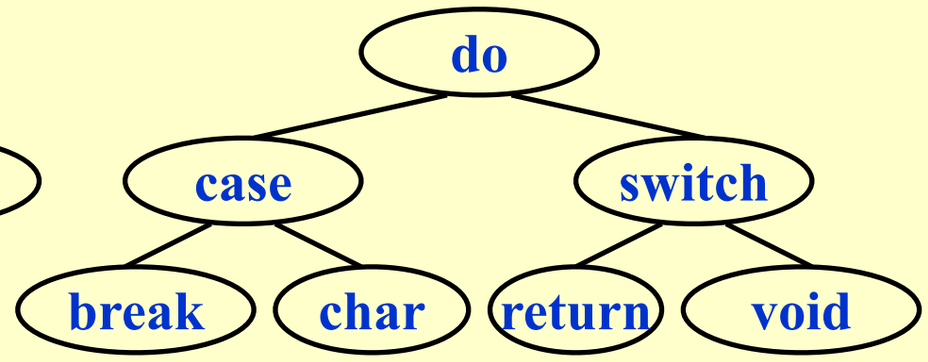
word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

Greedy Method



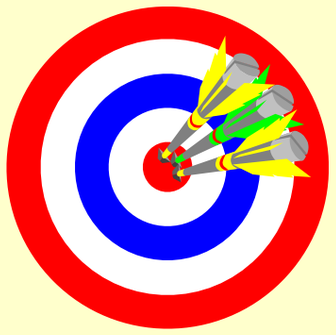
Time = 2.43

AVL Method



3. Optimal Binary Search Tree

—— The best for static searching (without insertion and deletion)

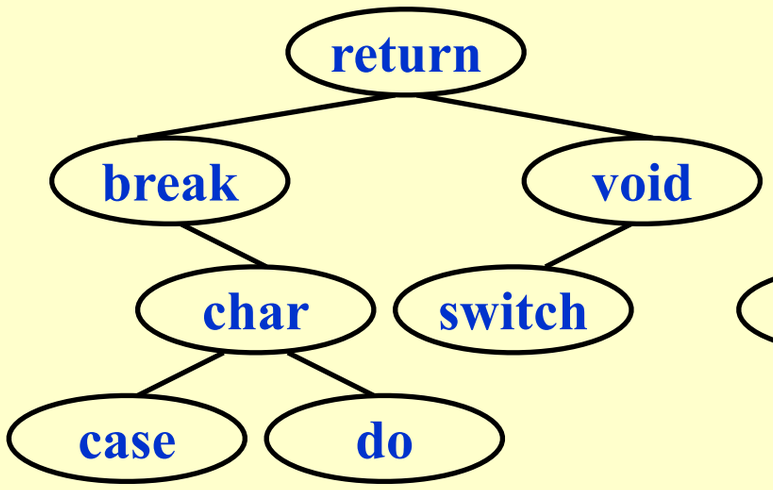


Given N words $w_1 < w_2 < \ldots\ldots < w_N$, and the probability of searching for each w_i is p_i. Arrange these words in a binary search tree in a way that minimize the expected total access time. $T(N) = \sum_{i=1}^{N} p_i \cdot (1 + d_i)$

〖Example〗 Given the following table of probabilities:

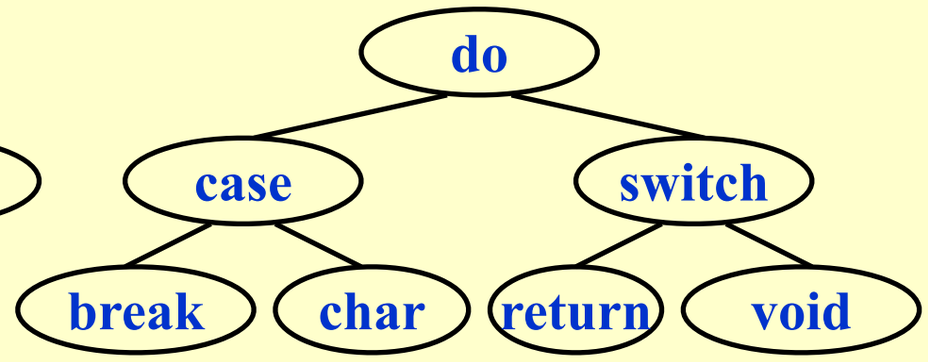
word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

Greedy Method



Time = 2.43

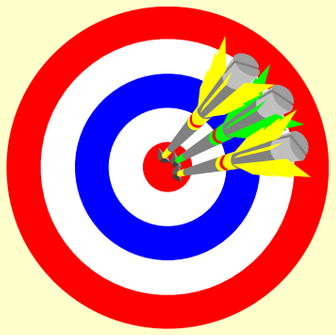
AVL Method



Time = 2.70

3. Optimal Binary Search Tree

—— The best for static searching (without insertion and deletion)

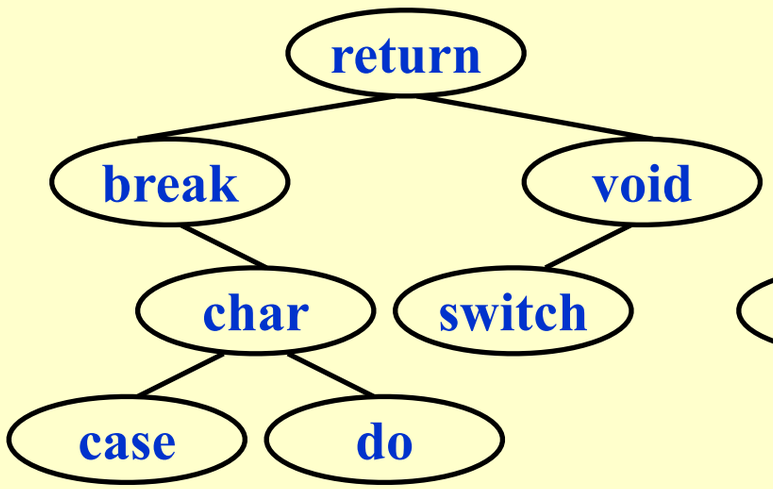


Given N words $w_1 < w_2 < \ldots\ldots < w_N$, and the probability of searching for each w_i is p_i. Arrange these words in a binary search tree in a way that minimize the expected total access time. $T(N) = \sum_{i=1}^{N} p_i \cdot (1 + d_i)$

〖Example〗 Given the following table of probabilities:

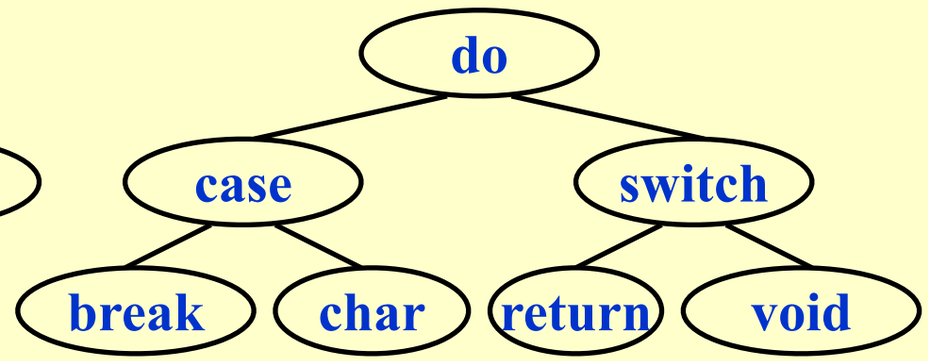
word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

Greedy Method



Time = 2.43

AVL Method

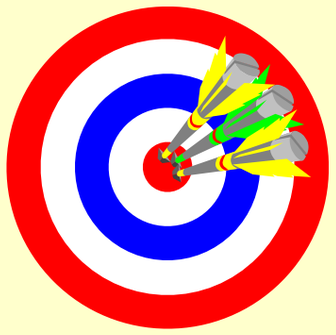


Time = 2.70

Optimal Tree

3. Optimal Binary Search Tree

—— The best for static searching (without insertion and deletion)

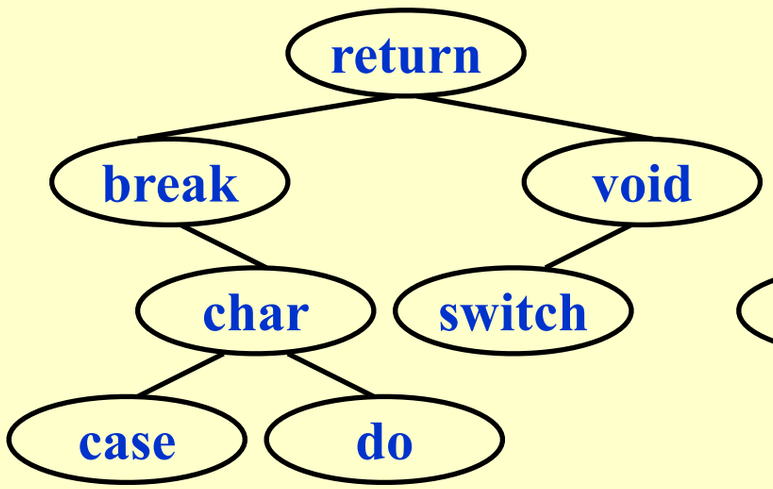


Given N words $w_1 < w_2 < \ldots\ldots < w_N$, and the probability of searching for each w_i is p_i. Arrange these words in a binary search tree in a way that minimize the expected total access time. $T(N) = \sum_{i=1}^{N} p_i \cdot (1 + d_i)$

〖Example〗 Given the following table of probabilities:

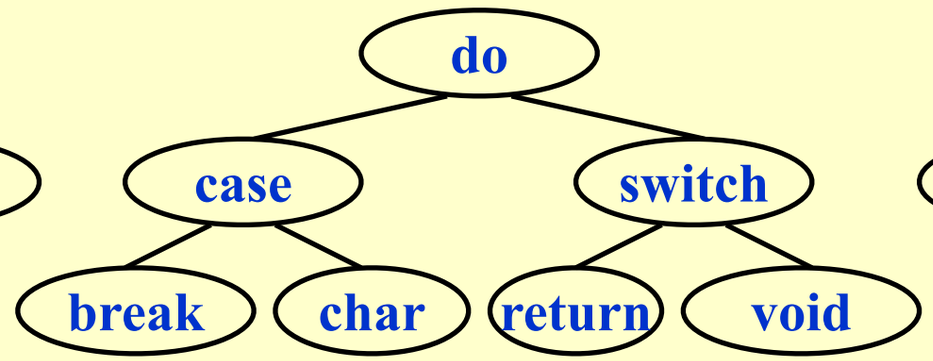
word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

Greedy Method



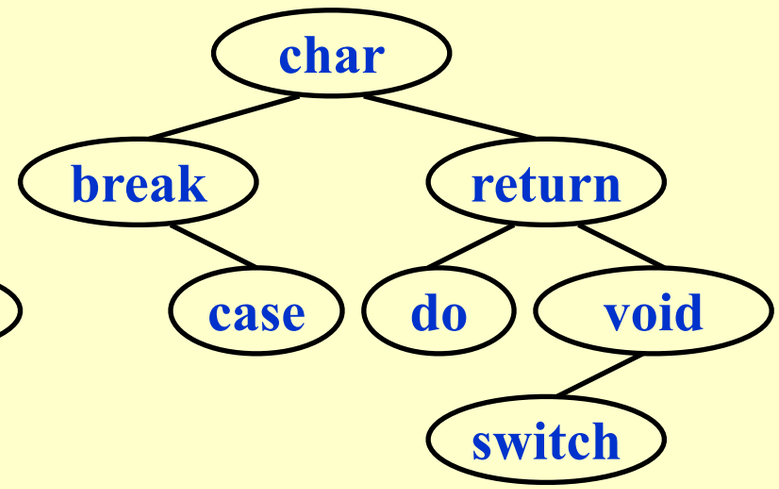
Time = 2.43

AVL Method



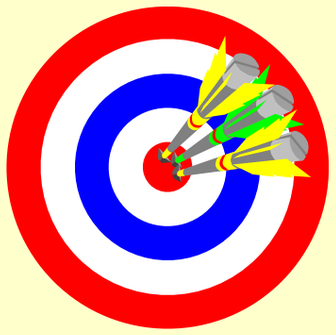
Time = 2.70

Optimal Tree



3. Optimal Binary Search Tree

—— The best for static searching (without insertion and deletion)

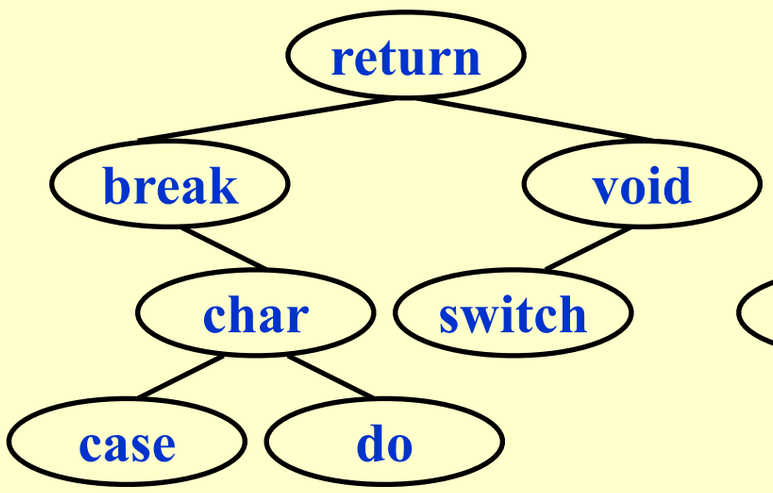


Given N words $w_1 < w_2 < \ldots\ldots < w_N$, and the probability of searching for each w_i is p_i. Arrange these words in a binary search tree in a way that minimize the expected total access time. $T(N) = \sum_{i=1}^{N} p_i \cdot (1 + d_i)$

〖Example〗 Given the following table of probabilities:

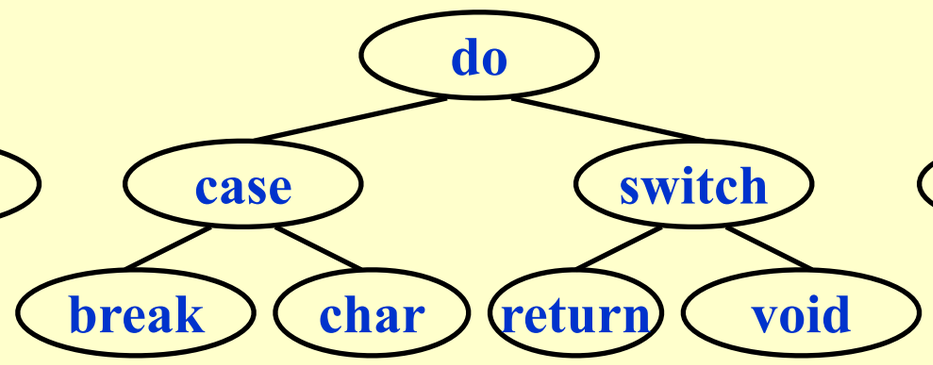
word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

Greedy Method



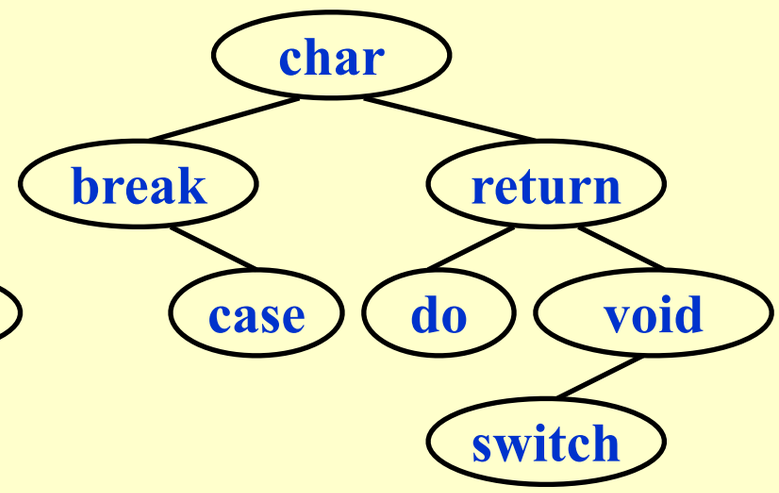
Time = 2.43

AVL Method



Time = 2.70

Optimal Tree



Time = 2.15

T_{ij} ::= OBST for $w_i, \ldots\ldots, w_j$ $(i < j)$

T_{ij} ::= OBST for $w_i, \ldots\ldots, w_j$ $(i < j)$

c_{ij} ::= cost of T_{ij} ($c_{ii} = 0$ if $j < i$)

T_{ij} ::= OBST for $w_i, \ldots\ldots, w_j$ $(i < j)$

c_{ij} ::= cost of T_{ij} ($c_{ii} = 0$ if $j < i$)

r_{ij} ::= root of T_{ij}

T_{ij} ::= OBST for $w_i, \ldots\ldots, w_j$ $(i < j)$

c_{ij} ::= cost of T_{ij} $(c_{ii} = 0$ if $j < i)$

r_{ij} ::= root of T_{ij}

w_{ij} ::= weight of $T_{ij} = \sum_{k=i}^{j} p_k$ $(w_{ii} = p_i)$

T_{ij} ::= OBST for $w_i, \ldots\ldots, w_j$ $(i < j)$

c_{ij} ::= cost of T_{ij} $(c_{ii} = 0$ if $j < i)$

r_{ij} ::= root of T_{ij}

w_{ij} ::= weight of $T_{ij} = \sum_{k=i}^{j} p_k$ $(w_{ii} = p_i)$

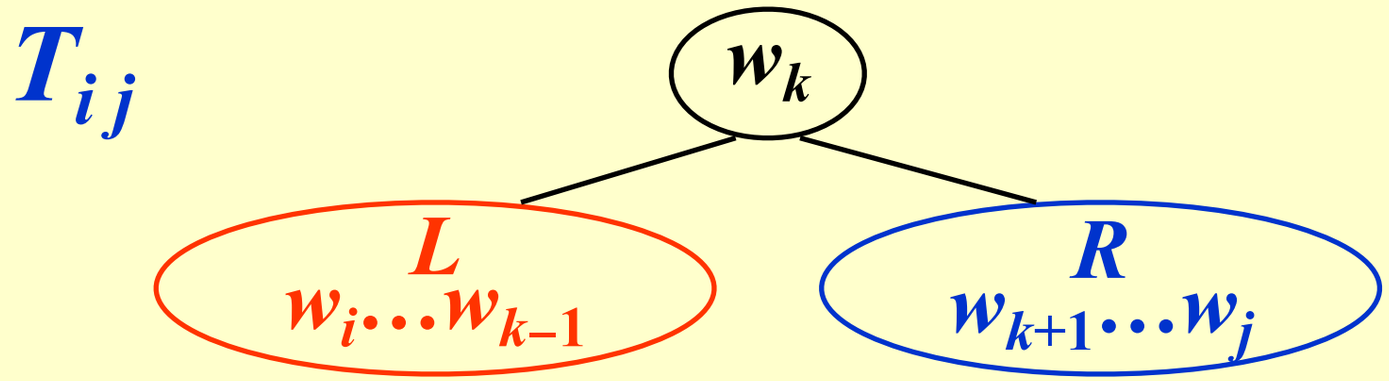
T_{1N} with root r_{1N}, weight w_{1N}, and cost c_{1N}.

T_{ij} ::= OBST for $w_i, \ldots\ldots, w_j$ $(i < j)$

c_{ij} ::= cost of T_{ij} ($c_{ii} = 0$ if $j < i$)

r_{ij} ::= root of T_{ij}

w_{ij} ::= weight of T_{ij} $= \sum_{k=i}^{j} p_k$ ($w_{ii} = p_i$)



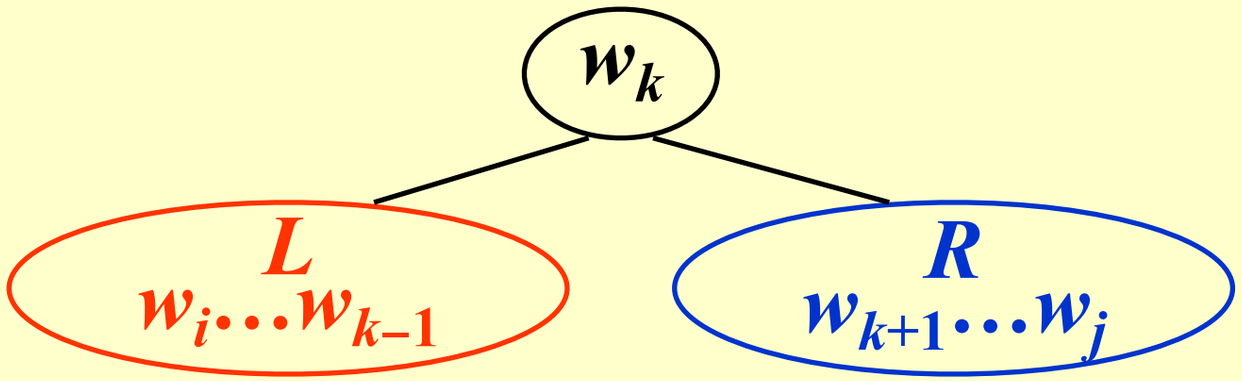
T_{ij} ::= OBST for $w_i, \ldots\ldots, w_j$ $(i < j)$

c_{ij} ::= cost of T_{ij} ($c_{ii} = 0$ if $j < i$)

r_{ij} ::= root of T_{ij}

w_{ij} ::= weight of $T_{ij} = \sum_{k=i}^{j} p_k$ ($w_{ii} = p_i$)

T_{ij}



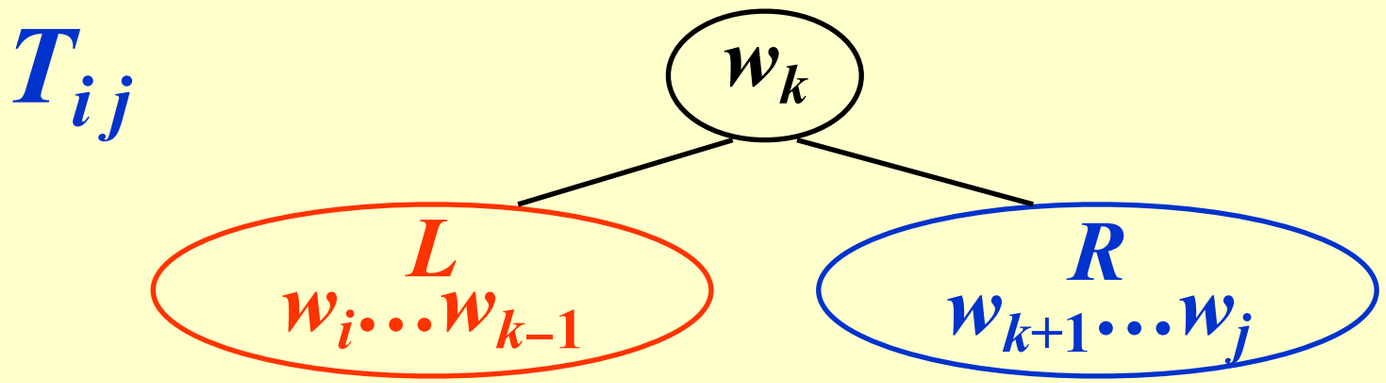
$c_{ij} =$?

T_{ij} ::= OBST for $w_i, \ldots\ldots, w_j$ $(i < j)$

c_{ij} ::= cost of T_{ij} ($c_{ii} = 0$ if $j < i$)

r_{ij} ::= root of T_{ij}

w_{ij} ::= weight of $T_{ij} = \sum_{k=i}^{j} p_k$ ($w_{ii} = p_i$)



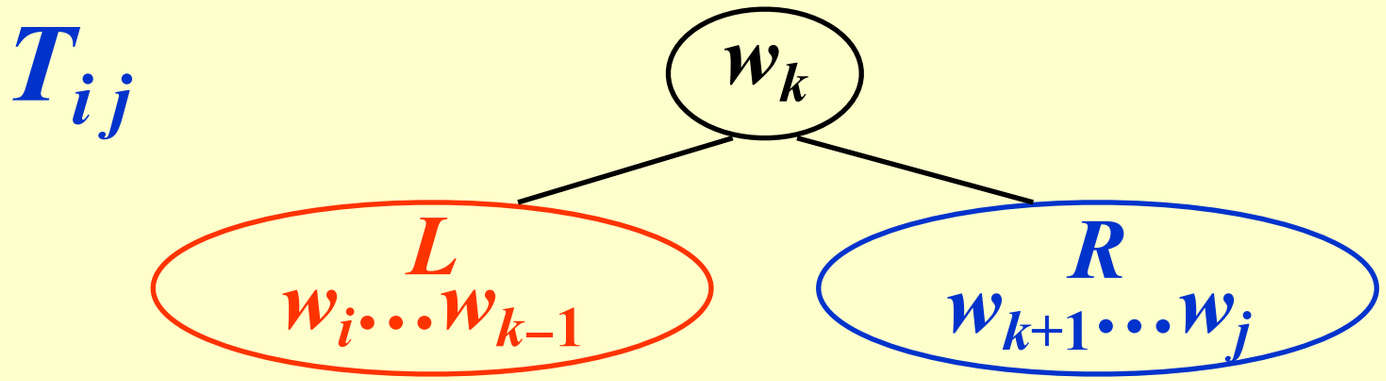
$c_{ij} = p_k + \text{cost}(L) + \text{cost}(R)$

T_{ij} ::= OBST for $w_i, \ldots\ldots, w_j$ $(i < j)$

c_{ij} ::= cost of T_{ij} ($c_{ii} = 0$ if $j < i$)

r_{ij} ::= root of T_{ij}

w_{ij} ::= weight of T_{ij} $= \sum_{k=i}^{j} p_k$ ($w_{ii} = p_i$)



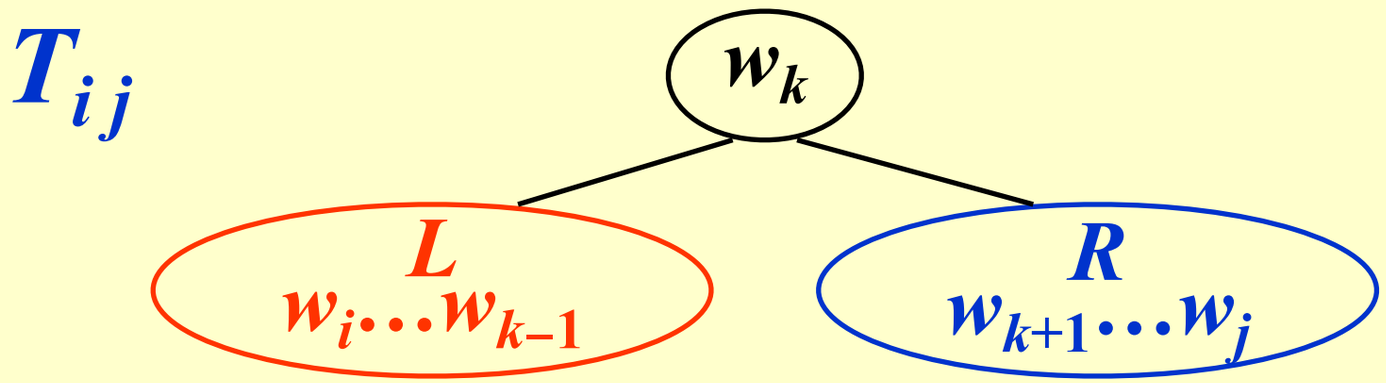
$c_{ij} = p_k$ + cost(L) + cost(R) + weight(L) + weight(R)

T_{ij} ::= OBST for $w_i, \ldots\ldots, w_j$ $(i < j)$

c_{ij} ::= cost of T_{ij} ($c_{ii} = 0$ if $j < i$)

r_{ij} ::= root of T_{ij}

w_{ij} ::= weight of $T_{ij} = \sum_{k=i}^{j} p_k$ ($w_{ii} = p_i$)



$c_{ij} = p_k + \text{cost}(L) + \text{cost}(R) + \text{weight}(L) + \text{weight}(R)$

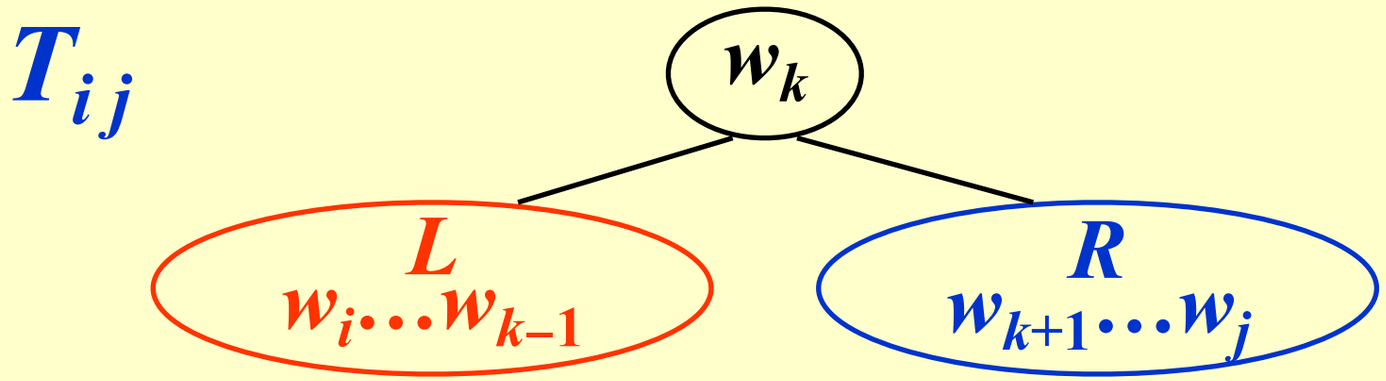
$= p_k + c_{i,k-1} + c_{k+1,j} + w_{i,k-1} + w_{k+1,j} = w_{ij} + c_{i,k-1} + c_{k+1,j}$

T_{ij} ::= OBST for $w_i, \ldots\ldots, w_j$ $(i < j)$

c_{ij} ::= cost of T_{ij} ($c_{ii} = 0$ if $j < i$)

r_{ij} ::= root of T_{ij}

w_{ij} ::= weight of $T_{ij} = \sum_{k=i}^{j} p_k$ ($w_{ii} = p_i$)



$c_{ij} = p_k + \text{cost}(L) + \text{cost}(R) + \text{weight}(L) + \text{weight}(R)$

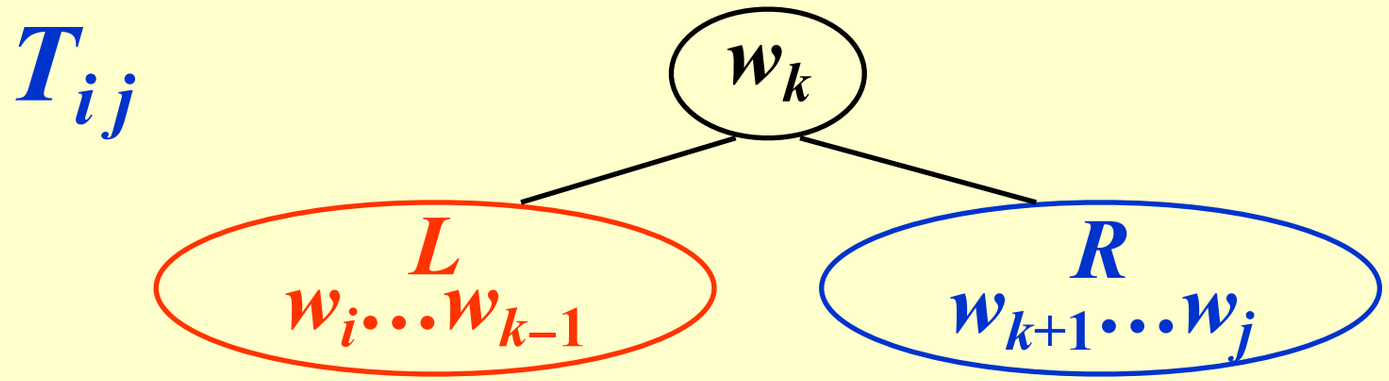
$= p_k + c_{i,k-1} + c_{k+1,j} + w_{i,k-1} + w_{k+1,j} = w_{ij} + c_{i,k-1} + c_{k+1,j}$

T_{ij} ::= OBST for $w_i, \ldots\ldots, w_j$ $(i < j)$

c_{ij} ::= cost of T_{ij} ($c_{ii} = 0$ if $j < i$)

r_{ij} ::= root of T_{ij}

w_{ij} ::= weight of $T_{ij} = \sum_{k=i}^{j} p_k$ ($w_{ii} = p_i$)



$c_{ij} = p_k + \text{cost}(L) + \text{cost}(R) + \text{weight}(L) + \text{weight}(R)$

$= p_k + c_{i,k-1} + c_{k+1,j} + w_{i,k-1} + w_{k+1,j} = w_{ij} + c_{i,k-1} + c_{k+1,j}$

T_{ij} is optimal $\Rightarrow$ $r_{ij} = k$ is such that $c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$

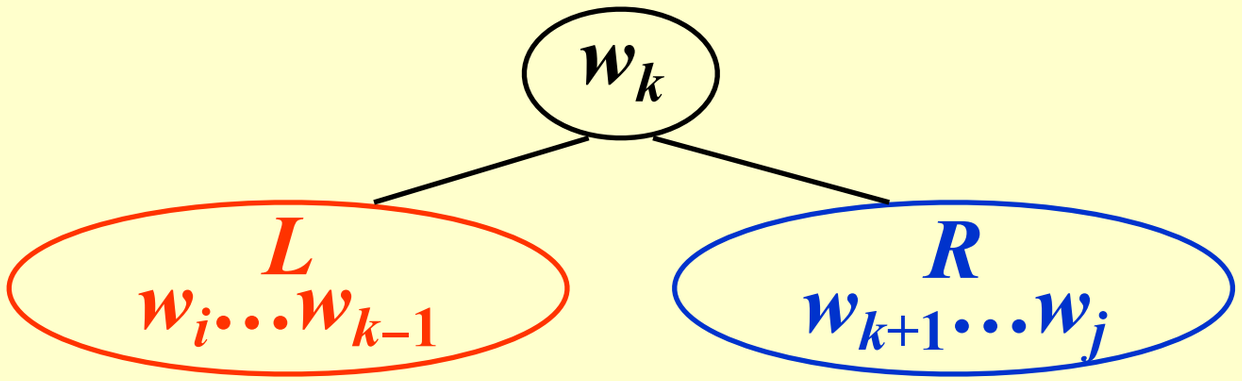
T_{ij} ::= OBST for $w_i, \ldots\ldots, w_j$ $(i < j)$

c_{ij} ::= cost of T_{ij} $(c_{ii} = 0$ if $j < i)$

r_{ij} ::= root of T_{ij}

w_{ij} ::= weight of $T_{ij} = \sum_{k=i}^{j} p_k$ $(w_{ii} = p_i)$

T_{ij}



$c_{ij} = p_k + \text{cost}(L) + \text{cost}(R) + \text{weight}(L) + \text{weight}(R)$

$= p_k + c_{i,k-1} + c_{k+1,j} + w_{i,k-1} + w_{k+1,j} = w_{ij} + c_{i,k-1} + c_{k+1,j}$

T_{ij} is optimal $\Rightarrow$ $r_{ij} = k$ is such that $c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case													
0.58	break												

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case				case.. char									
0.58	break			0.56	char								

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do									
0.58	break	0.56	char	0.30	char								

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return							
0.58	break	0.56	char	0.30	char	0.35	return						

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch					
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return				

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char													
1.02	case												

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do											
1.02	case	0.66	char										

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return									
1.02	case	0.66	char	0.80	return								

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch							
1.02	case	0.66	char	0.80	return	0.39	return						

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do													
1.17	case												

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return											
1.17	case	1.21	char										

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return		char.. switch									
1.17	case	1.21	char	0.84	return								

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return		char.. switch		do.. void							
1.17	case	1.21	char	0.84	return	0.57	return						

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return		char.. switch		do.. void							
1.17	case	1.21	char	0.84	return	0.57	return						
break.. return													
1.83	char												

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return		char.. switch		do.. void							
1.17	case	1.21	char	0.84	return	0.57	return						
break.. return		case.. switch											
1.83	char	1.27	char										

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return		char.. switch		do.. void							
1.17	case	1.21	char	0.84	return	0.57	return						
break.. return		case.. switch		char.. void									
1.83	char	1.27	char	1.02	return								

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return		char.. switch		do.. void							
1.17	case	1.21	char	0.84	return	0.57	return						
break.. return		case.. switch		char.. void									
1.83	char	1.27	char	1.02	return								
break.. switch													
1.89	char												

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return		char.. switch		do.. void							
1.17	case	1.21	char	0.84	return	0.57	return						
break.. return		case.. switch		char.. void									
1.83	char	1.27	char	1.02	return								
break.. switch		case.. void											
1.89	char	1.53	char										

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return		char.. switch		do.. void							
1.17	case	1.21	char	0.84	return	0.57	return						
break.. return		case.. switch		char.. void									
1.83	char	1.27	char	1.02	return								
break.. switch		case.. void											
1.89	char	1.53	char										
break.. void													
2.15	char												

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return		char.. switch		do.. void							
1.17	case	1.21	char	0.84	return	0.57	return						
break.. return		case.. switch		char.. void									
1.83	char	1.27	char	1.02	return								
break.. switch		case.. void											
1.89	char	1.53	char										
break.. void													
2.15	char												

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

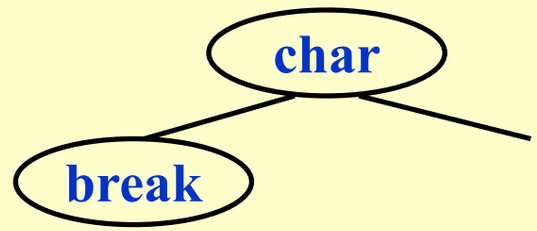
word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return		char.. switch		do.. void							
1.17	case	1.21	char	0.84	return	0.57	return						
break.. return		case.. switch		char.. void									
1.83	char	1.27	char	1.02	return								
break.. switch		case.. void											
1.89	char	1.53	char										
break.. void													
2.15	char												

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

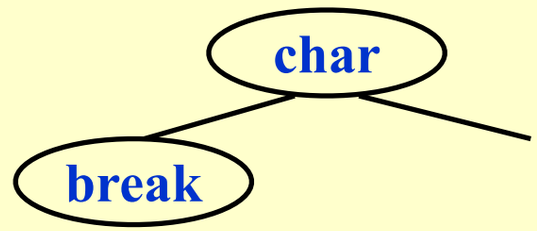
break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return		char.. switch		do.. void							
1.17	case	1.21	char	0.84	return	0.57	return						
break.. return		case.. switch		char.. void									
1.83	char	1.27	char	1.02	return								
break.. switch		case.. void											
1.89	char	1.53	char										
break.. void													
2.15	char												



$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

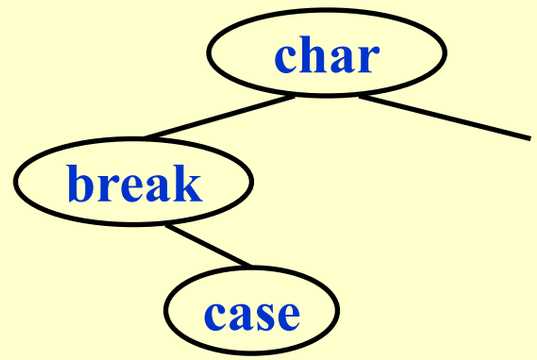
break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return		char.. switch		do.. void							
1.17	case	1.21	char	0.84	return	0.57	return						
break.. return		case.. switch		char.. void									
1.83	char	1.27	char	1.02	return								
break.. switch		case.. void											
1.89	char	1.53	char										
break.. void													
2.15	char												



$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

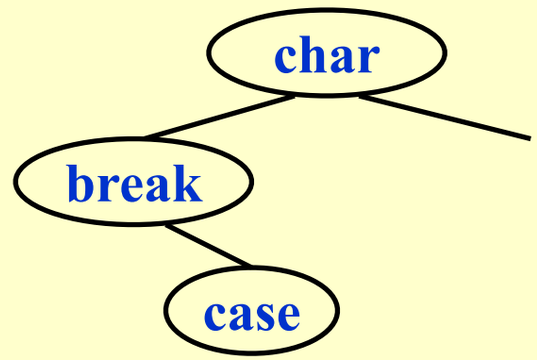
break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return		char.. switch		do.. void							
1.17	case	1.21	char	0.84	return	0.57	return						
break.. return		case.. switch		char.. void									
1.83	char	1.27	char	1.02	return								
break.. switch		case.. void											
1.89	char	1.53	char										
break.. void													
2.15	char												



$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

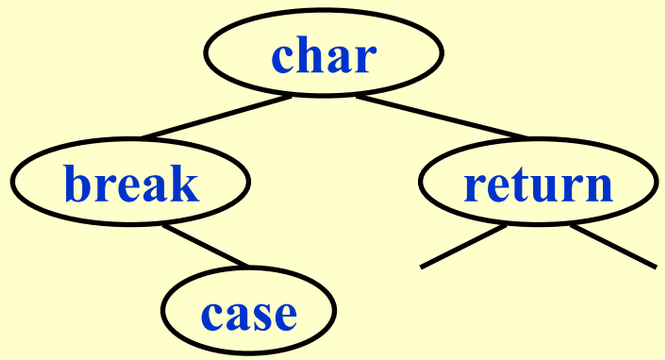
break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return		char.. switch		do.. void							
1.17	case	1.21	char	0.84	return	0.57	return						
break.. return		case.. switch		char.. void									
1.83	char	1.27	char	1.02	return								
break.. switch		case.. void											
1.89	char	1.53	char										
break.. void													
2.15	char												



$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

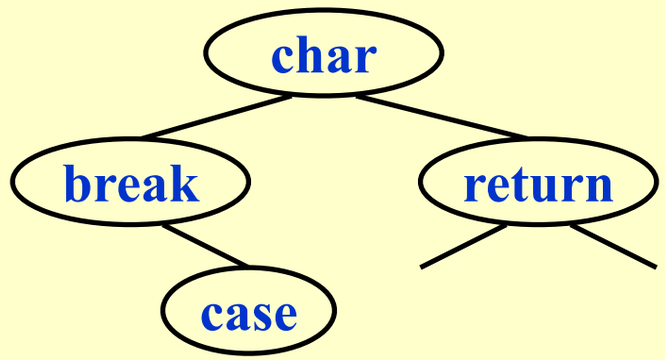
break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return		char.. switch		do.. void							
1.17	case	1.21	char	0.84	return	0.57	return						
break.. return		case.. switch		char.. void									
1.83	char	1.27	char	1.02	return								
break.. switch		case.. void											
1.89	char	1.53	char										
break.. void													
2.15	char												



$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

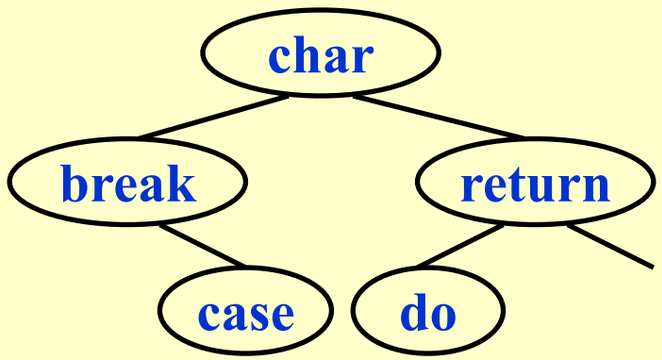
break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return		char.. switch		do.. void							
1.17	case	1.21	char	0.84	return	0.57	return						
break.. return		case.. switch		char.. void									
1.83	char	1.27	char	1.02	return								
break.. switch		case.. void											
1.89	char	1.53	char										
break.. void													
2.15	char												



$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

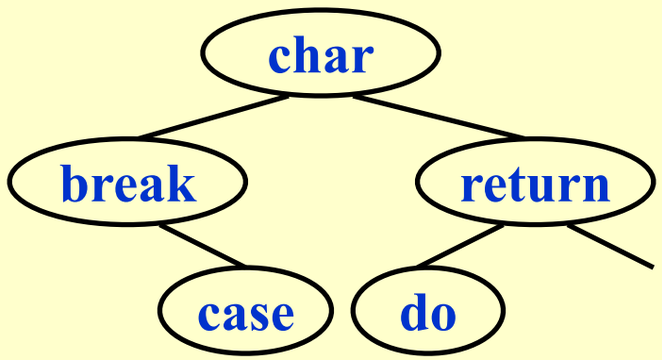
break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return		char.. switch		do.. void							
1.17	case	1.21	char	0.84	return	0.57	return						
break.. return		case.. switch		char.. void									
1.83	char	1.27	char	1.02	return								
break.. switch		case.. void											
1.89	char	1.53	char										
break.. void													
2.15	char												



$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

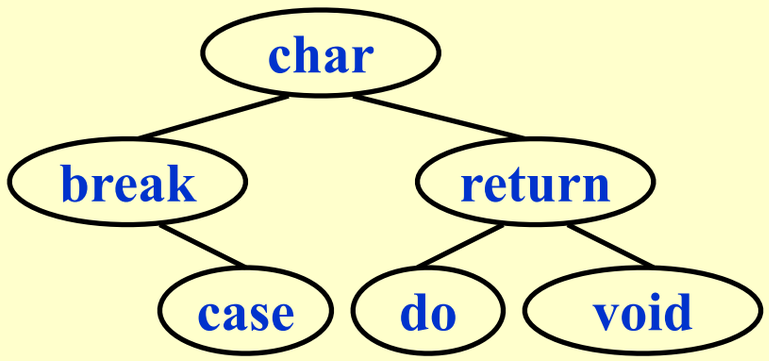
break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return		char.. switch		do.. void							
1.17	case	1.21	char	0.84	return	0.57	return						
break.. return		case.. switch		char.. void									
1.83	char	1.27	char	1.02	return								
break.. switch		case.. void											
1.89	char	1.53	char										
break.. void													
2.15	char												



$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

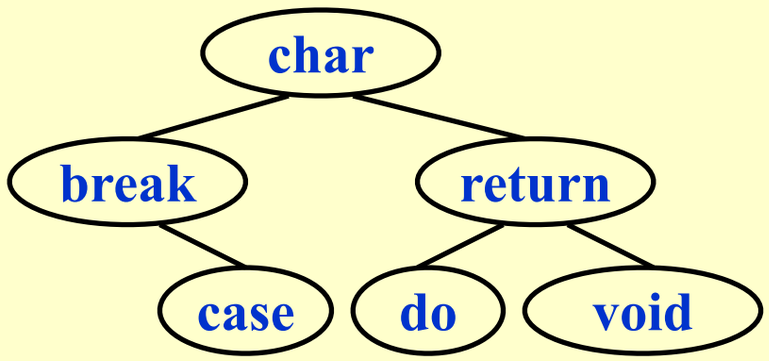
break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return		char.. switch		do.. void							
1.17	case	1.21	char	0.84	return	0.57	return						
break.. return		case.. switch		char.. void									
1.83	char	1.27	char	1.02	return								
break.. switch		case.. void											
1.89	char	1.53	char										
break.. void													
2.15	char												



$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

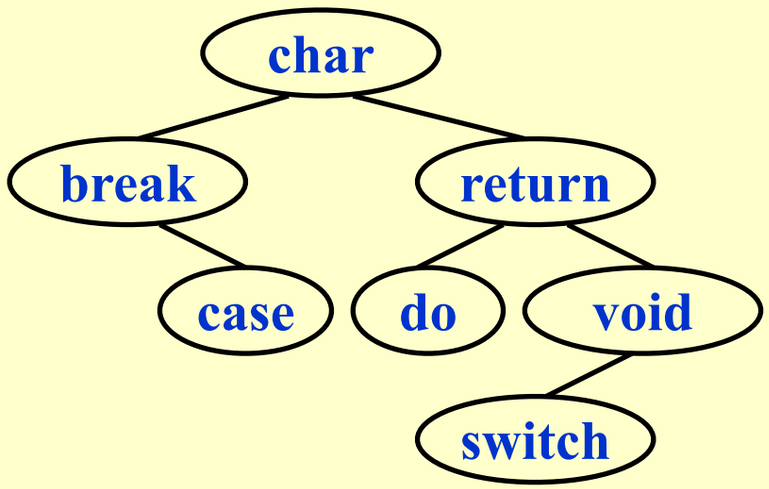
break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return		char.. switch		do.. void							
1.17	case	1.21	char	0.84	return	0.57	return						
break.. return		case.. switch		char.. void									
1.83	char	1.27	char	1.02	return								
break.. switch		case.. void											
1.89	char	1.53	char										
break.. void													
2.15	char												



$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

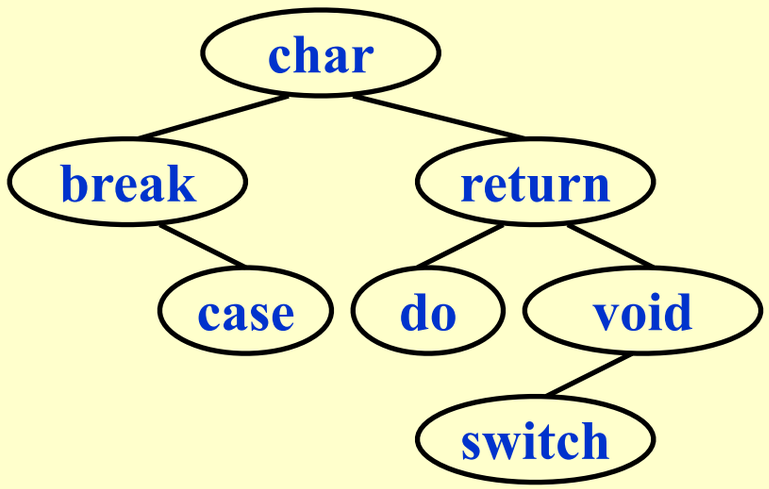
break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return		char.. switch		do.. void							
1.17	case	1.21	char	0.84	return	0.57	return						
break.. return		case.. switch		char.. void									
1.83	char	1.27	char	1.02	return								
break.. switch		case.. void											
1.89	char	1.53	char										
break.. void													
2.15	char												



$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return		char.. switch		do.. void							
1.17	case	1.21	char	0.84	return	0.57	return						
break.. return		case.. switch		char.. void									
1.83	char	1.27	char	1.02	return								
break.. switch		case.. void											
1.89	char	1.53	char										
break.. void													
2.15	char												

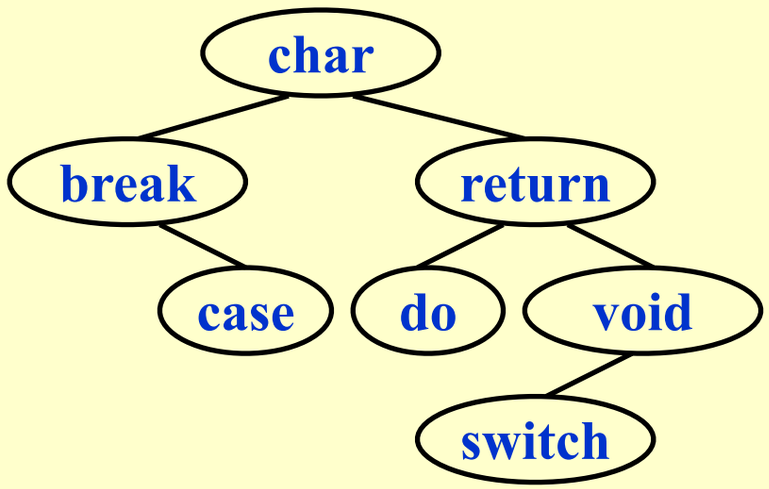


$$T(N) = O(N^3)$$

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return		char.. switch		do.. void							
1.17	case	1.21	char	0.84	return	0.57	return						
break.. return		case.. switch		char.. void									
1.83	char	1.27	char	1.02	return								
break.. switch		case.. void											
1.89	char	1.53	char										
break.. void													
2.15	char												



$$T(N) = O(N^3)$$

Please read Weiss 10.33 on p.419 for an $O(N^2)$ algorithm.

4. All-Pairs Shortest Path

For all pairs of v_i and v_j ($i \neq j$), find the shortest path between.

4. All-Pairs Shortest Path

For all pairs of v_i and v_j ($i \neq j$), find the shortest path between.

Method 1 **Use single-source algorithm for |V| times.**

4. All-Pairs Shortest Path

For all pairs of v_i and v_j ($i \neq j$), find the shortest path between.

Method 1 **Use single-source algorithm for |V| times.**

$T = O(|V|^3)$ **– works fast on sparse graph.**

4. All-Pairs Shortest Path

For all pairs of v_i and v_j ($i \neq j$), find the shortest path between.

Method 1 **Use single-source algorithm for |V| times.**

$T = O(|V|^3)$ **– works fast on sparse graph.**

Method 2 **Define**

$D^k[\,i\,][\,j\,] = \min\{$ length of path $i \rightarrow \{ l \leq k \} \rightarrow j \}$

and $D^{-1}[\,i\,][\,j\,] = Cost[\,i\,][\,j\,]$. **Then the length of the shortest path from i to j is**

4. All-Pairs Shortest Path

For all pairs of v_i and v_j ($i \neq j$), find the shortest path between.

Method 1 **Use single-source algorithm for |V| times.**

$T = O(|V|^3)$ **– works fast on sparse graph.**

Method 2 **Define**

$D^k[\,i\,][\,j\,] = \min\{$ length of path $i \rightarrow \{ l \leq k \} \rightarrow j \}$

and $D^{-1}[\,i\,][\,j\,] = Cost[\,i\,][\,j\,]$. **Then the length of the shortest path from** i **to** j **is** $D^{N-1}[\,i\,][\,j\,]$.

4. All-Pairs Shortest Path

For all pairs of v_i and v_j ($i \neq j$), find the shortest path between.

Method 1 Use **single-source algorithm** for |V| times.

$T = O(|V|^3)$ – works fast on sparse graph.

Method 2 Define

$D^k[\,i\,][\,j\,] = \min\{$ length of path $i \rightarrow \{ l \leq k \} \rightarrow j \}$

and $D^{-1}[\,i\,][\,j\,] = Cost[\,i\,][\,j\,]$. Then the length of the shortest path from i to j is $D^{N-1}[\,i\,][\,j\,]$.

Algorithm

Start from D^{-1} and successively generate D^0, D^1, ..., D^{N-1}. If D^{k-1} is done, then either

4. All-Pairs Shortest Path

For all pairs of v_i and v_j ($i \neq j$), find the shortest path between.

Method 1 Use **single-source algorithm** for |V| times.

$T = O(|V|^3)$ – works fast on sparse graph.

Method 2 Define

$D^k[\,i\,][\,j\,] = \min\{$ length of path $i \rightarrow \{ l \leq k \} \rightarrow j \}$

and $D^{-1}[\,i\,][\,j\,] = Cost[\,i\,][\,j\,]$. Then the length of the shortest path from i to j is $D^{N-1}[\,i\,][\,j\,]$.

Algorithm

Start from D^{-1} and successively generate D^0, D^1, ..., D^{N-1}. If D^{k-1} is done, then either

① $k \notin$ the shortest path $i \rightarrow \{ l \leq k \} \rightarrow j \Rightarrow D^k = D^{k-1}$;

or

4. All-Pairs Shortest Path

For all pairs of v_i and v_j ($i \neq j$), find the shortest path between.

Method 1 Use **single-source algorithm** for |V| times.

$T = O(|V|^3)$ – works fast on sparse graph.

Method 2 Define

$D^k[\,i\,][\,j\,] = \min\{$ length of path $i \rightarrow \{\, l \leq k \,\} \rightarrow j \,\}$

and $D^{-1}[\,i\,][\,j\,] = \text{Cost}[\,i\,][\,j\,]$. Then the length of the shortest path from i to j is $D^{N-1}[\,i\,][\,j\,]$.

Algorithm

Start from D^{-1} and successively generate D^0, D^1, ..., D^{N-1}. If D^{k-1} is done, then either

① $k \notin$ the shortest path $i \rightarrow \{\, l \leq k \,\} \rightarrow j \;\Rightarrow\; D^k = D^{k-1}$;

or

② $k \in$ the shortest path $i \rightarrow \{\, l \leq k \,\} \rightarrow j$

$=$ { the S.P. from i to k } $\cup$ {the S.P. from k to j }

$\Rightarrow\; D^k[\,i\,][\,j\,] = D^{k-1}[\,i\,][\,k\,] + D^{k-1}[\,k\,][\,j\,]$

4. All-Pairs Shortest Path

For all pairs of v_i and v_j ($i \neq j$), find the shortest path between.

Method 1 Use **single-source algorithm** for |V| times.

$T = O(|V|^3)$ – works fast on sparse graph.

Method 2 Define

$D^k[\,i\,][\,j\,] = \min\{$ length of path $i \rightarrow \{\, l \leq k \,\} \rightarrow j \,\}$

and $D^{-1}[\,i\,][\,j\,] = \text{Cost}[\,i\,][\,j\,]$. Then the length of the shortest path from i to j is $D^{N-1}[\,i\,][\,j\,]$.

Algorithm

Start from D^{-1} and successively generate D^0, D^1, ..., D^{N-1}. If D^{k-1} is done, then either

① $k \notin$ the shortest path $i \rightarrow \{\, l \leq k \,\} \rightarrow j \;\Rightarrow\; D^k = D^{k-1}$;

or

② $k \in$ the shortest path $i \rightarrow \{\, l \leq k \,\} \rightarrow j$

$=$ { the S.P. from i to k } $\cup$ {the S.P. from k to j }

$\Rightarrow\; D^k[\,i\,][\,j\,] = D^{k-1}[\,i\,][\,k\,] + D^{k-1}[\,k\,][\,j\,]$

$$\therefore\quad D^k[i][j] = \min\{D^{k-1}[i][j],\, D^{k-1}[i][k] + D^{k-1}[k][j]\},\ k \geq 0$$

```
/* A[ ] contains the adjacency matrix with A[ i ][ i ] = 0 */
/* D[ ] contains the values of the shortest path */
/* N is the number of vertices */
/* A negative cycle exists iff D[ i ][ i ] < 0 */
void AllPairs( TwoDimArray A, TwoDimArray D, int N )
{   int  i, j, k;
    for ( i = 0; i < N; i++ )  /* Initialize D */
        for( j = 0; j < N; j++ )
                D[ i ][ j ] = A[ i ][ j ];
    for( k = 0; k < N; k++ )  /* add one vertex k into the path */
        for( i = 0; i < N; i++ )
                for( j = 0; j < N; j++ )
                    if( D[ i ][ k ] + D[ k ][ j ] < D[ i ][ j ] )
                            /* Update shortest path */
                             D[ i ][ j ] = D[ i ][ k ] + D[ k ][ j ];
}
```

```
/* A[ ] contains the adjacency matrix with A[ i ][ i ] = 0 */
/* D[ ] contains the values of the shortest path */
/* N is the number of vertices */
/* A negative cycle exists iff D[ i ][ i ] < 0 */
void AllPairs( TwoDimArray A, TwoDimArray D, int N )
{   int  i, j, k;
    for ( i = 0; i < N; i++ )  /* Initialize D */
        for( j = 0; j < N; j++ )
                D[ i ][ j ] = A[ i ][ j ];
    for( k = 0; k < N; k++ )  /* add one vertex k into the path */
        for( i = 0; i < N; i++ )
                for( j = 0; j < N; j++ )
                    if( D[ i ][ k ] + D[ k ][ j ] < D[ i ][ j ] )
                            /* Update shortest path */
                            D[ i ][ j ] = D[ i ][ k ] + D[ k ][ j ];
}
```

$T(N) = O(N^3)$, but faster in a *dense* graph.

```
/* A[ ] contains the adjacency matrix with A[ i ][ i ] = 0 */
/* D[ ] contains the values of the shortest path */
/* N is the number of vertices */
/* A negative cycle exists iff D[ i ][ i ] < 0 */
void AllPairs( TwoDimArray A, TwoDimArray D, int N )
{   int  i, j, k;
    for ( i = 0; i < N; i++ )  /* Initialize D */
        for( j = 0; j < N; j++ )
                D[ i ][ j ] = A[ i ][ j ];
    for( k = 0; k < N; k++ )  /* add one vertex k into the path */
        for( i = 0; i < N; i++ )
                for( j = 0; j < N; j++ )
                    if( D[ i ][ k ] + D[ k ][ j ] < D[ i ][ j ] )
                            /* Update shortest path */
                            D[ i ][ j ] = D[ i ][ k ] + D[ k ][ j ];
}
```

$T(N) = O(N^3)$, but faster in a *dense* graph.

To record the paths please refer to Weiss Figure 10.53 on p.393

```
/* A[ ] contains the adjacency matrix with A[ i ][ i ] = 0 */
/* D[ ] contains the values of the shortest path */
/* N is the number of vertices */
/* A negative cycle exists iff D[ i ][ i ] < 0 */
void AllPairs( TwoDimArray A, TwoDimArray D, int N )
{   int  i, j, k;
    for ( i = 0; i < N; i++ )     /* Initialize D */
        for( j = 0; j < N; j++ )
                D[ i ][ j ] = A[ i ][ j ];
    for( k = 0; k < N; k++ )
        for( i = 0; i < N; i++ )
            for( j = 0; j < N; j++ )
                if( D[ i ][ k ] + D[ k ][ j ] < D[ i ][ j ] )
                    /* Update shortest path */
                    D[ i ][ j ] = D[ i ][ k ] + D[ k ][ j ];
}
```

Works if there are negative edge costs, but no negative-cost cycles.

$T(N) = O(N^3)$, but faster in a *dense* graph.

To record the paths please refer to Weiss Figure 10.53 on p.393

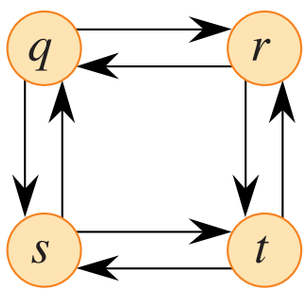


Figure 14.6 A directed graph showing that the problem of finding a longest simple path in an unweighted directed graph does not have optimal substructure. The path $q \to r \to t$ is a longest simple path from q to t, but the subpath $q \to r$ is not a longest simple path from q to r, nor is the subpath $r \to t$ a longest simple path from r to t.

It is essential to indeed verify whether the sub-structure optimal condition really holds.
Read CLRS (4th edition) pp. 386

How to design a DP method?

How to design a DP method?

☞ **Characterize an optimal solution**

How to design a DP method?

- ☞ Characterize an optimal solution
  - ☞ Recursively define the optimal values

How to design a DP method?

- ☞ **Characterize an optimal solution**
- ☞ **Recursively define the optimal values**
- ☞ **Compute the values in some order**

How to design a DP method?

- ☞ **Characterize an optimal solution**
- ☞ **Recursively define the optimal values**
- ☞ **Compute the values in some order**
- ☞ **Reconstruct the solving strategy**

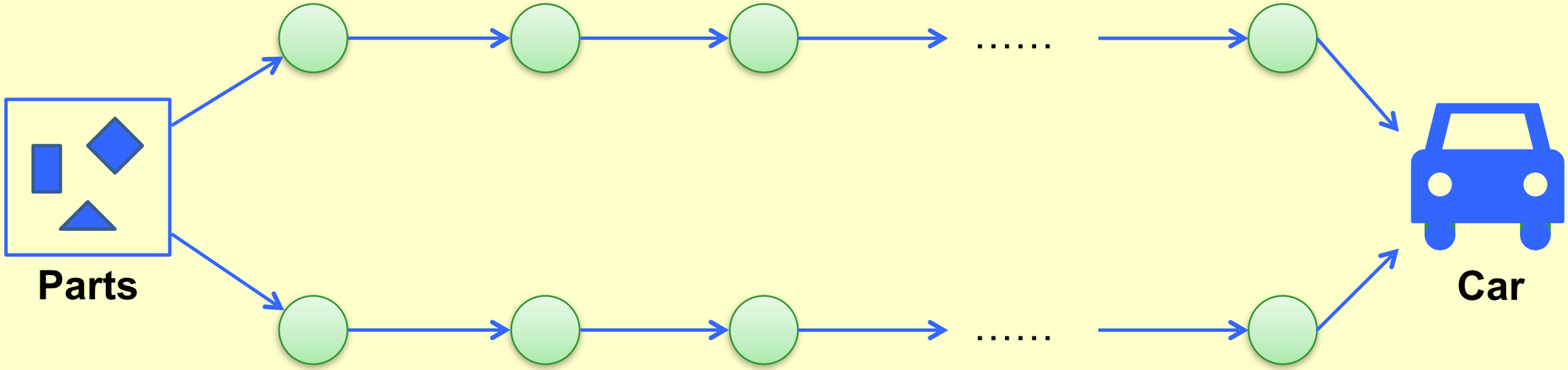
5. Product Assembly

5. Product Assembly

- **Two assembly lines for the same car**

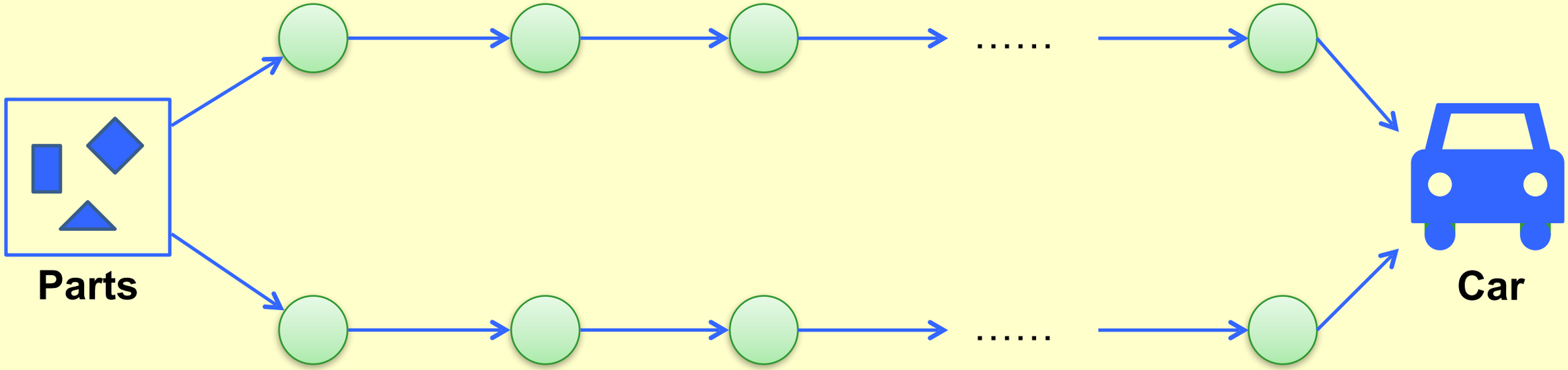
5. Product Assembly

- **Two assembly lines for the same car**



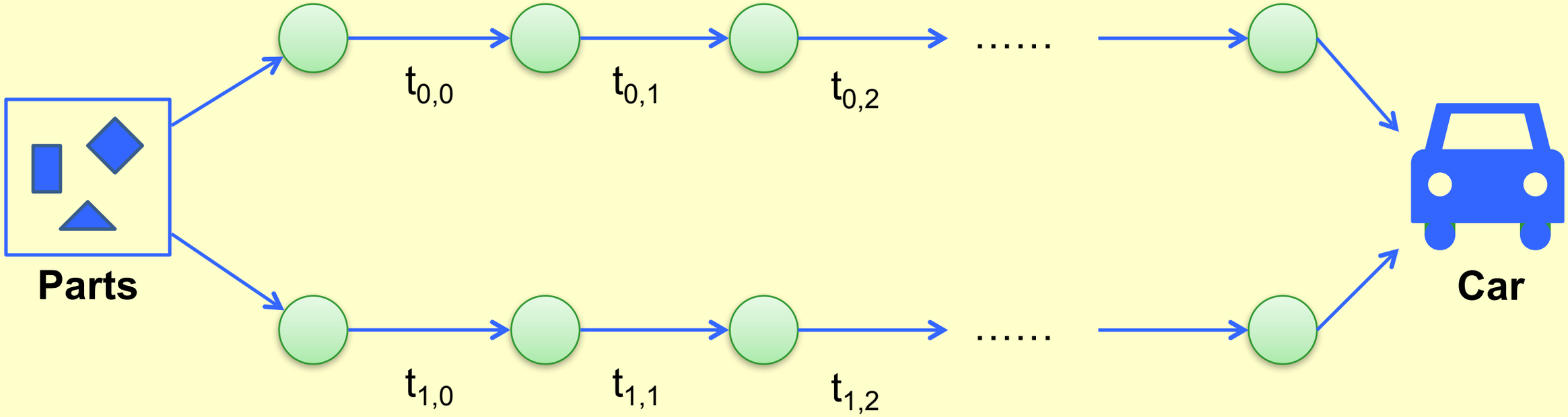
5. Product Assembly

- **Two assembly lines for the same car**
- **Different technology (time) for each stage**



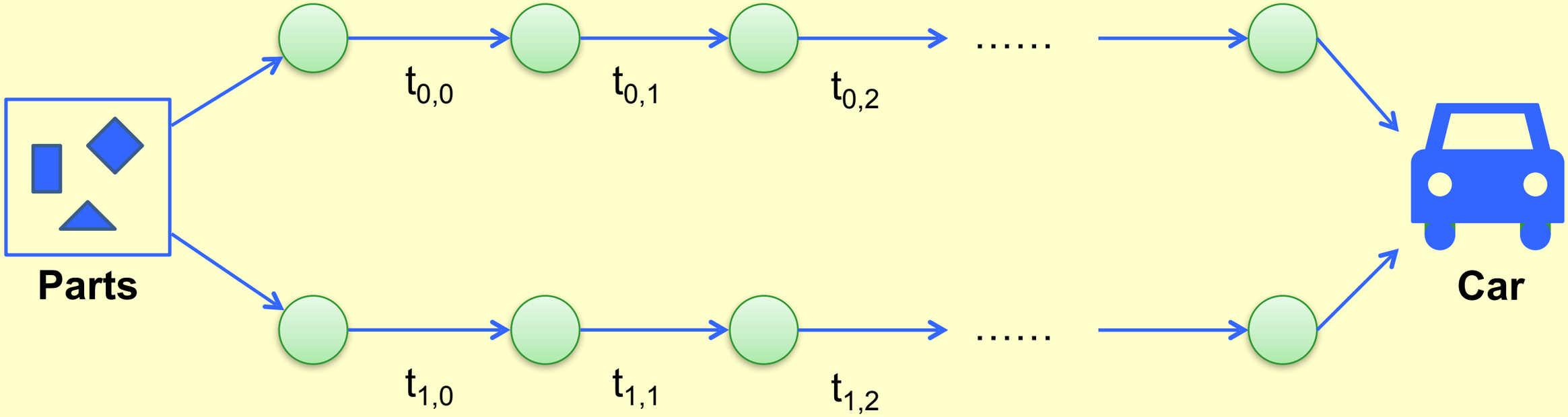
5. Product Assembly

- **Two assembly lines for the same car**
- **Different technology (time) for each stage**



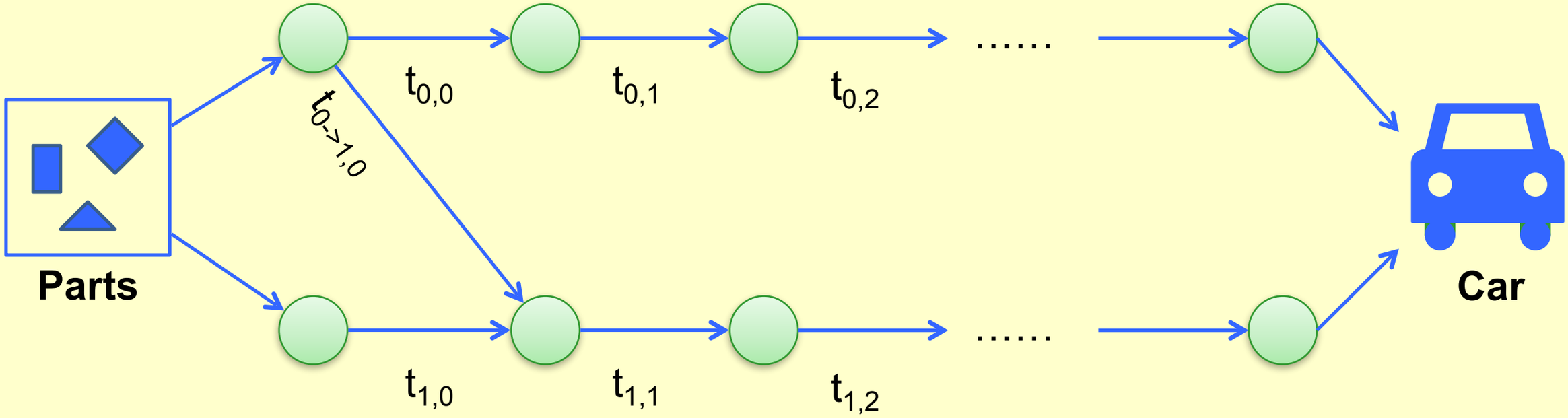
5. Product Assembly

- Two assembly lines for the same car
- Different technology (time) for each stage
- One can change lines between stages



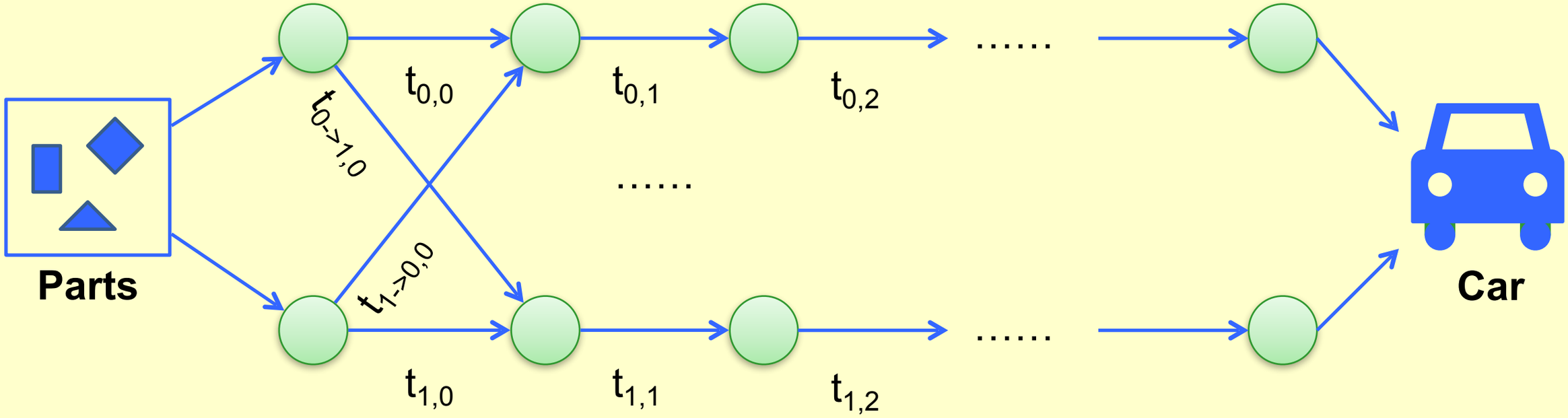
5. Product Assembly

- Two assembly lines for the same car
- Different technology (time) for each stage
- One can change lines between stages



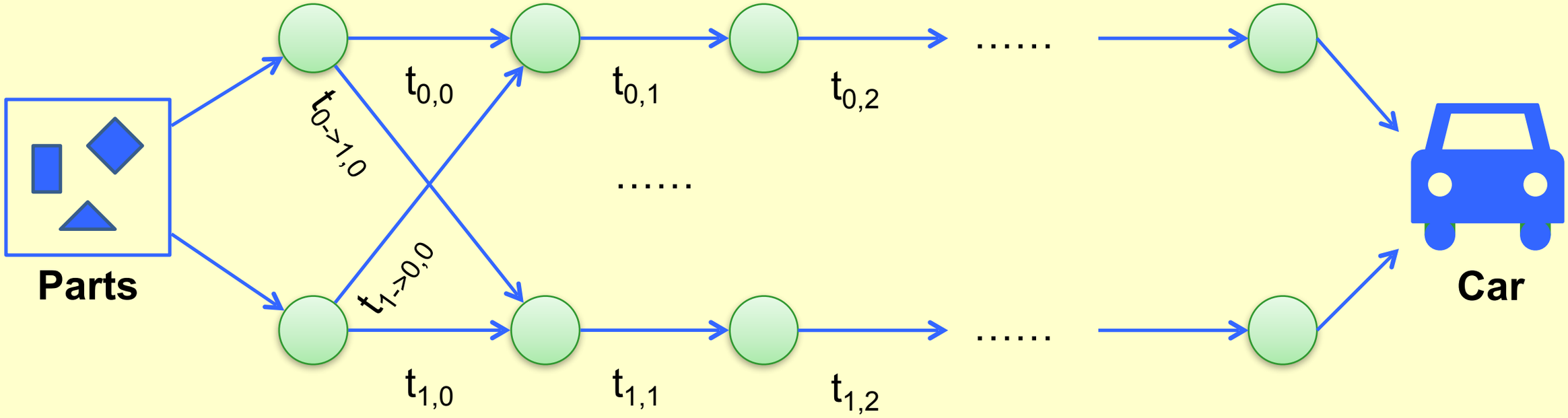
5. Product Assembly

- Two assembly lines for the same car
- Different technology (time) for each stage
- One can change lines between stages



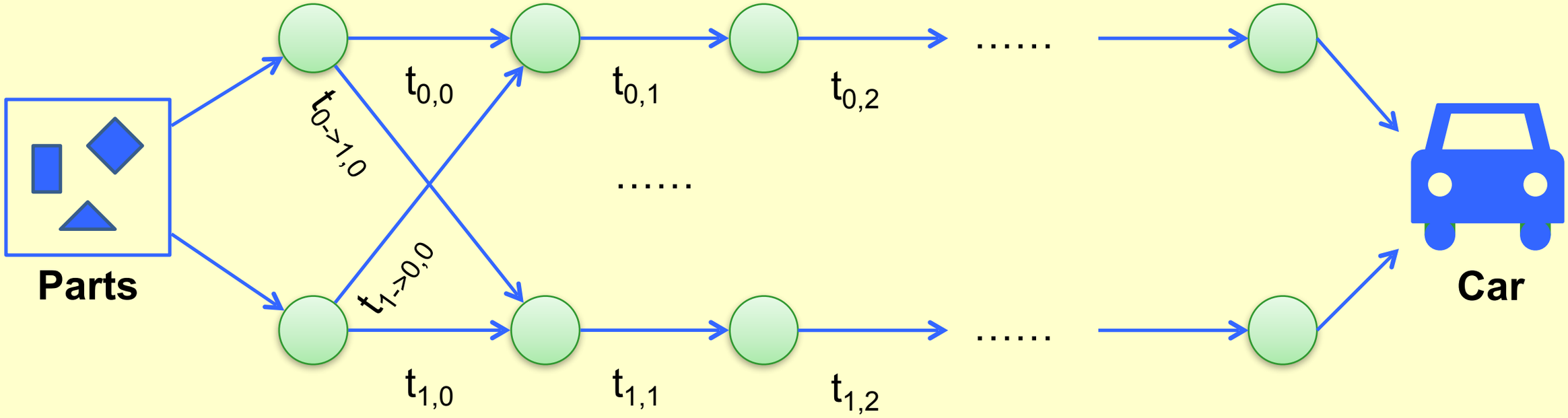
5. Product Assembly

- Two assembly lines for the same car
- Different technology (time) for each stage
- One can change lines between stages
- Minimize the total assembly time



5. Product Assembly

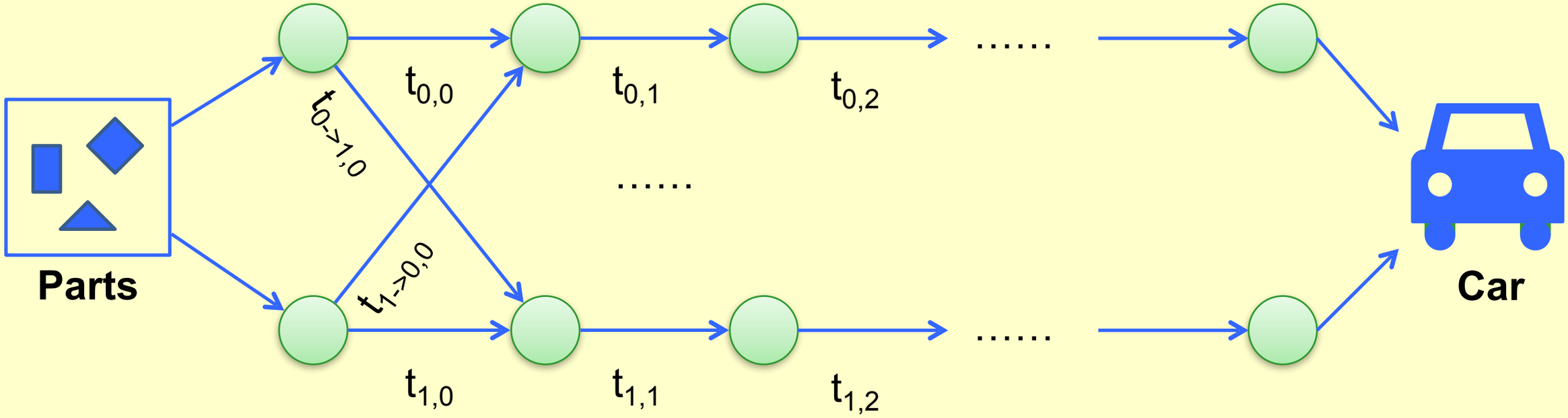
- Two assembly lines for the same car
- Different technology (time) for each stage
- One can change lines between stages
- Minimize the total assembly time



Exhaustive search gives O(2^N) time + O(N) space

5. Product Assembly

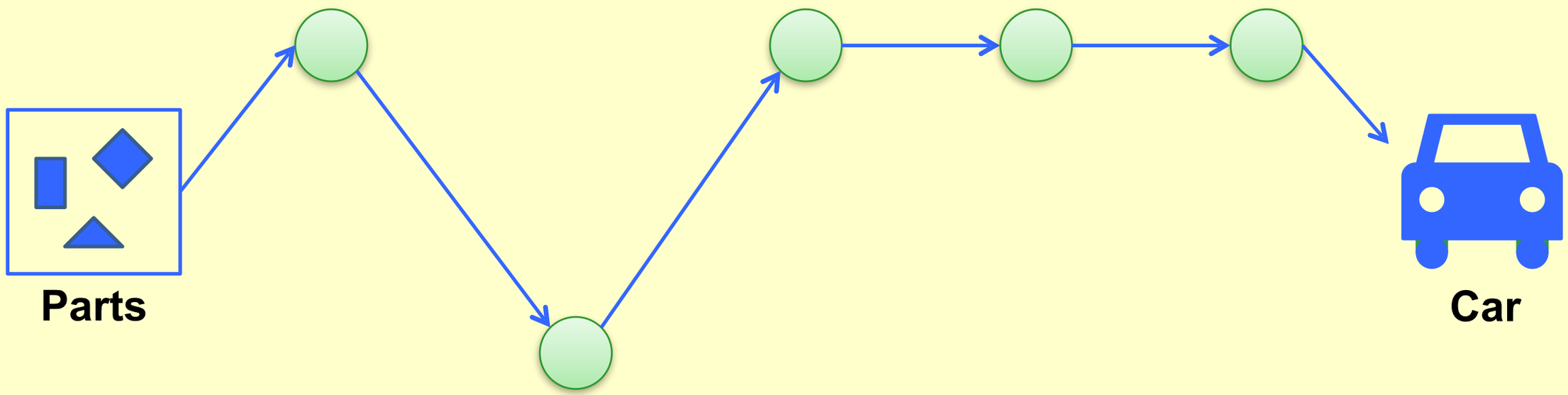
- Two assembly lines for the same car
- Different technology (time) for each stage
- One can change lines between stages
- Minimize the total assembly time



Exhaustive search gives O(2^N) time + O(N) space

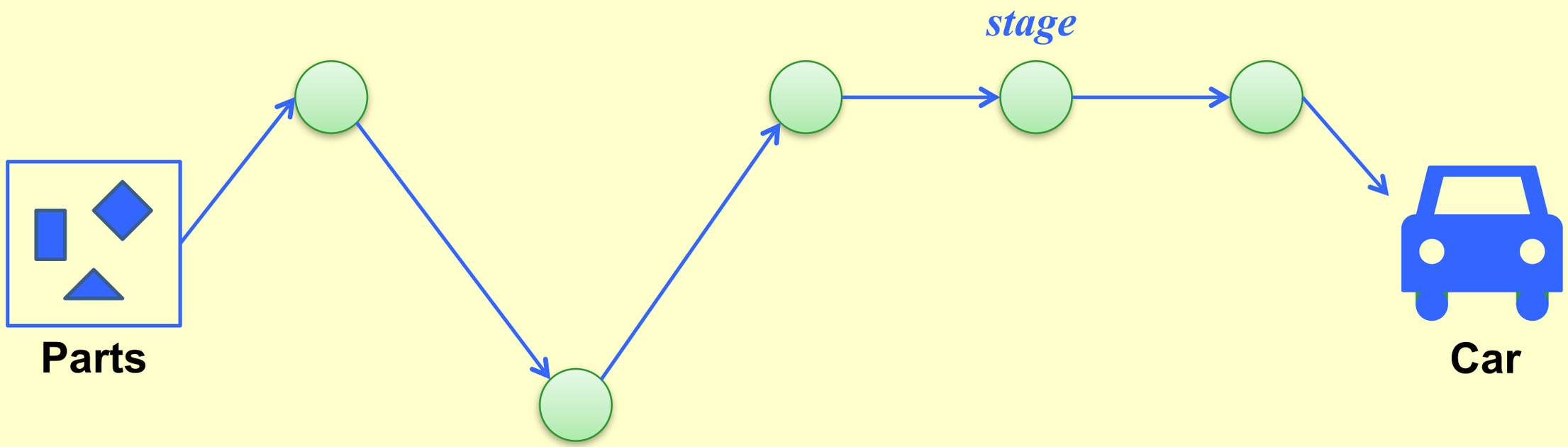
☞ Characterize an optimal solution

☞ Characterize an optimal solution



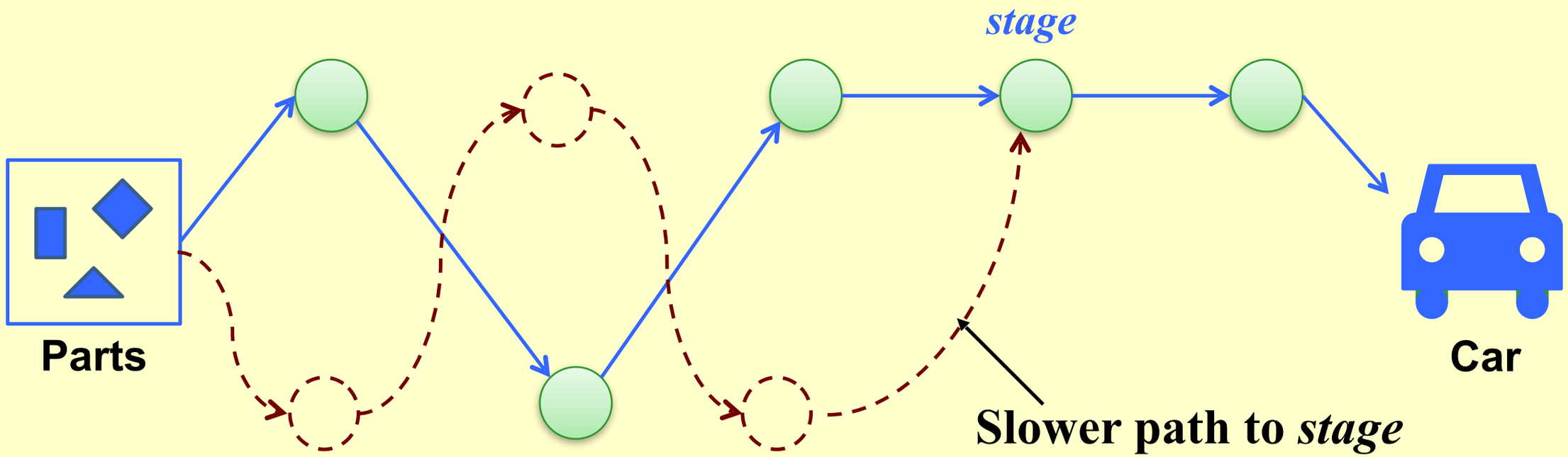
Optimal solution

☞ Characterize an optimal solution



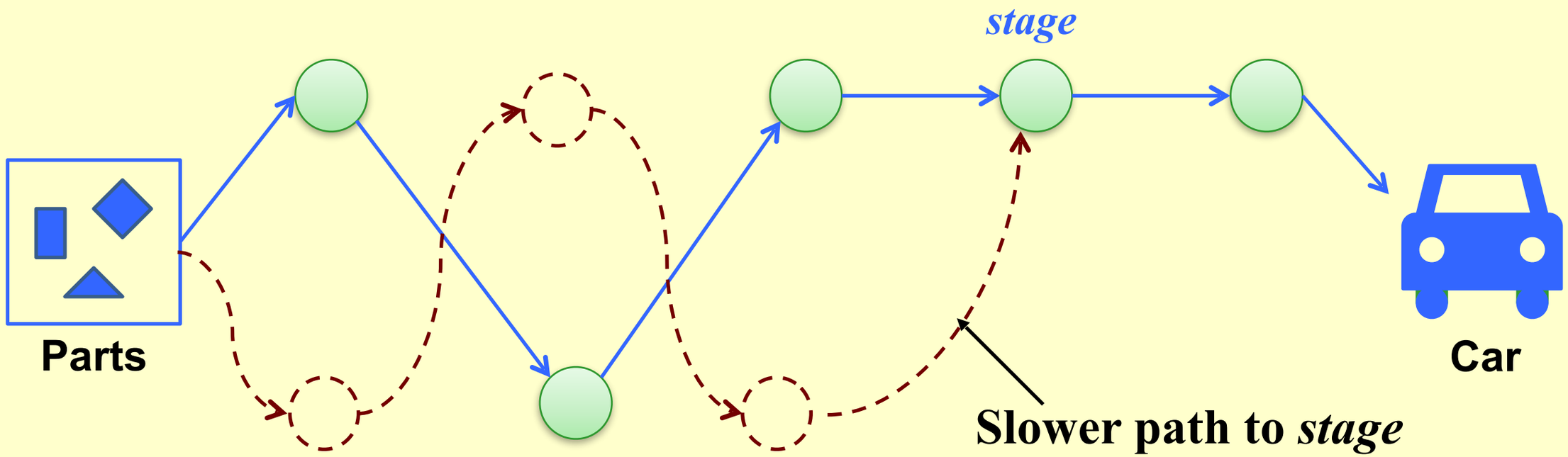
Optimal solution

☞ **Characterize an optimal solution**



Optimal solution

☞ Characterize an optimal solution



Optimal solution

☝ **An optimal solution contains an optimal solution of a sub-problem!**

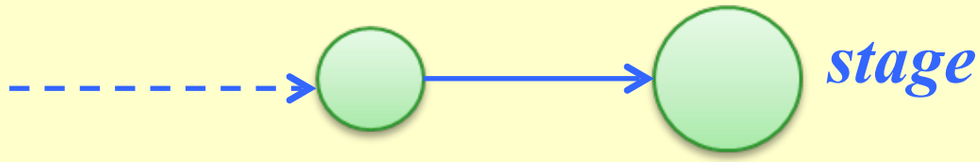
☞ **Recursively define the optimal values**

☞ Recursively define the optimal values

An optimal path to *stage* is based on an optimal path to *stage*–1

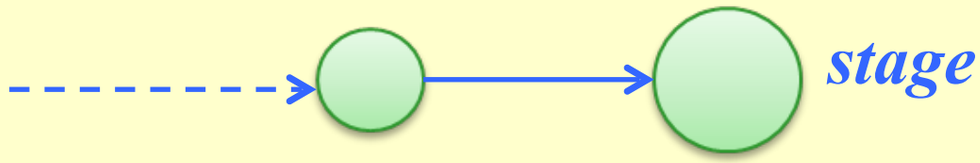
☞ Recursively define the optimal values

An optimal path to *stage* is based on an optimal path to *stage*–1



☞ Recursively define the optimal values

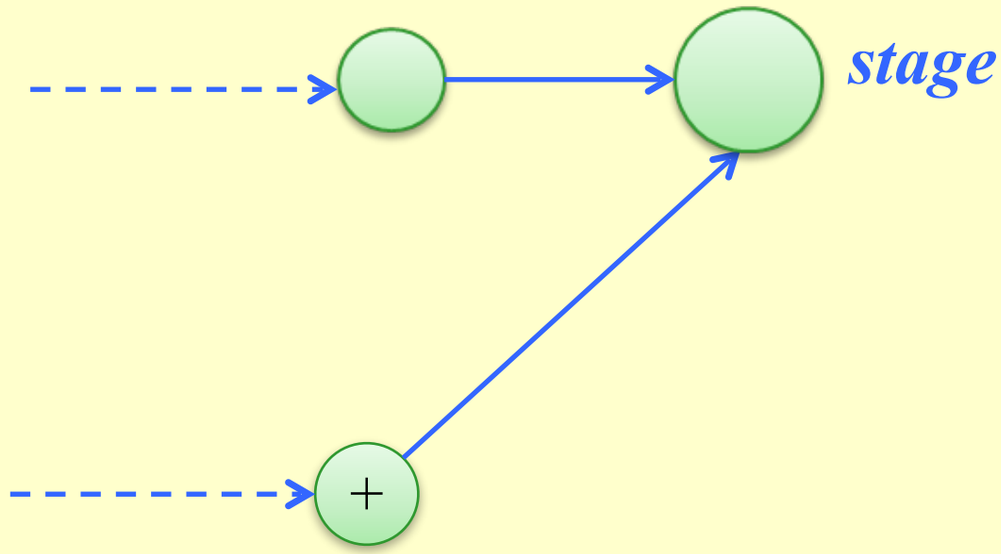
An optimal path to *stage* is based on an optimal path to *stage*–1




```
f[line][stage] =
    f[  line][stage-1] + t_process[  line][stage-1],
```

☞ Recursively define the optimal values

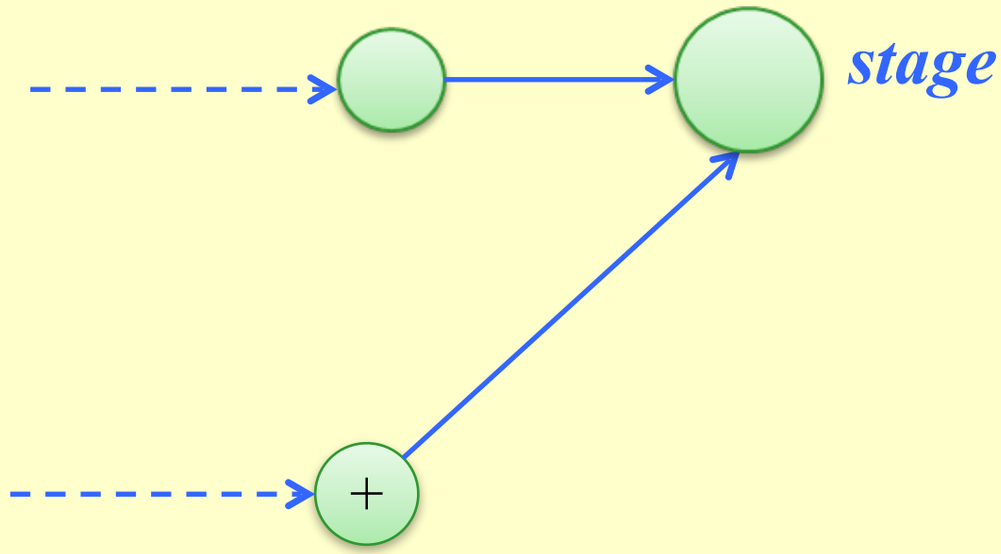
An optimal path to *stage* is based on an optimal path to *stage*–1




```
f[line][stage] =
    f[  line][stage-1] + t_process[  line][stage-1],
```

☞ Recursively define the optimal values

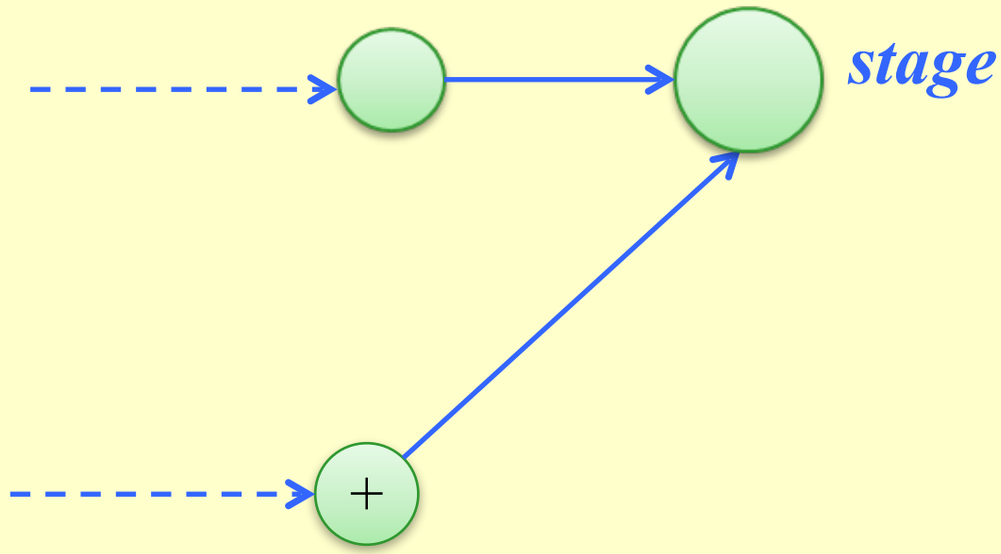
An optimal path to *stage* is based on an optimal path to *stage*–1




```
f[line][stage] =
    f[  line][stage-1] + t_process[  line][stage-1],
    f[1-line][stage-1] + t_transit[1-line][stage-1])
```

☞ Recursively define the optimal values

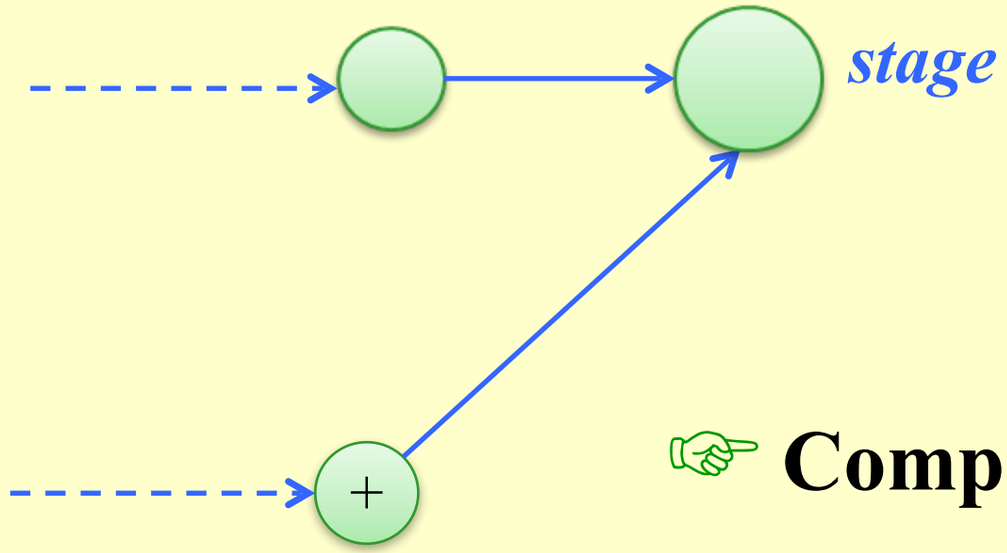
An optimal path to *stage* is based on an optimal path to *stage*–1




```
f[line][stage] = min(
    f[  line][stage-1] + t_process[  line][stage-1],
    f[1-line][stage-1] + t_transit[1-line][stage-1]);
```

☞ Recursively define the optimal values

An optimal path to *stage* is based on an optimal path to *stage*–1

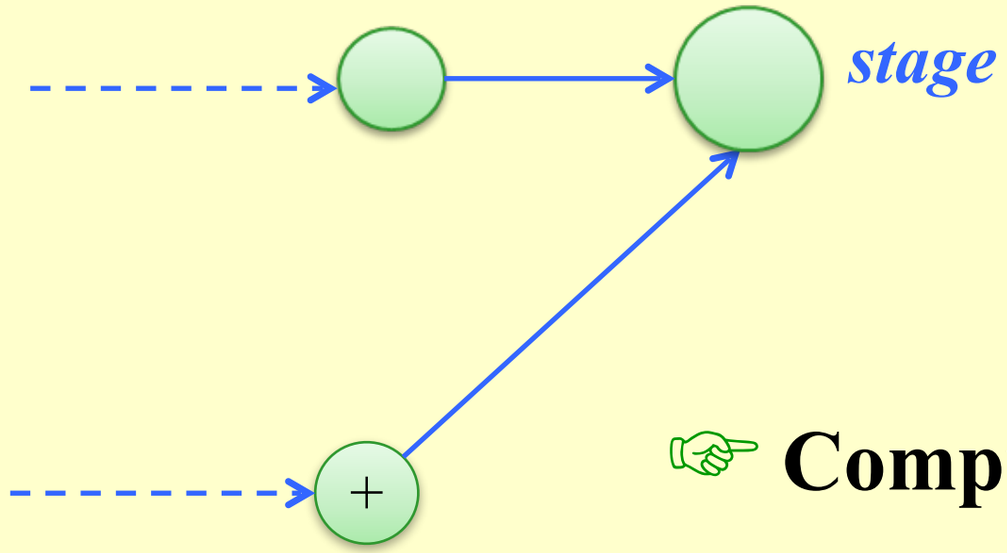


☞ Compute the values in some order

```
f[line][stage] = min(
    f[  line][stage-1] + t_process[  line][stage-1],
    f[1-line][stage-1] + t_transit[1-line][stage-1]);
```

☞ Recursively define the optimal values

An optimal path to *stage* is based on an optimal path to *stage*–1



☞ Compute the values in some order

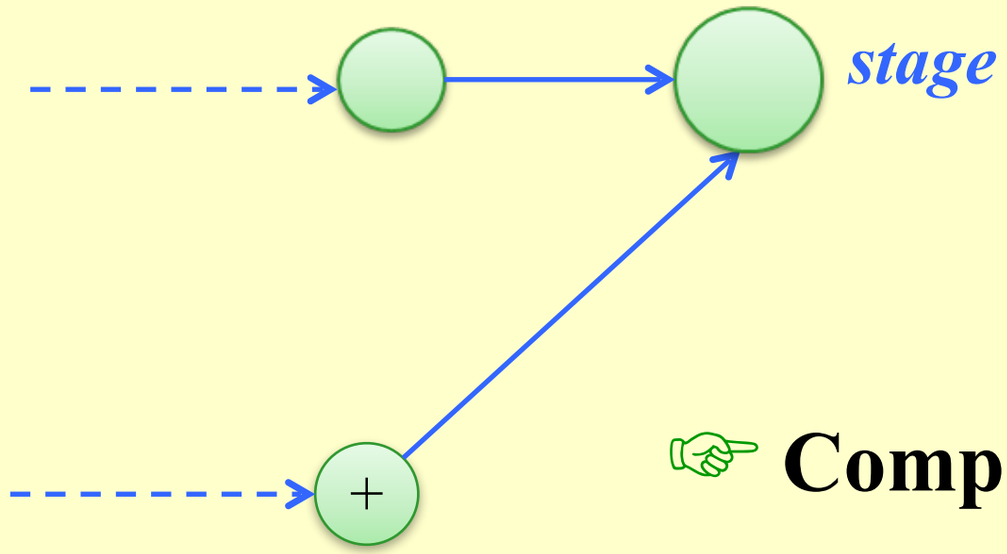
```
for (stage=1; stage<=n; stage++){

   f[line][stage] = min(
       f[  line][stage-1] + t_process[  line][stage-1],
       f[1-line][stage-1] + t_transit[1-line][stage-1]);

}
```

☞ Recursively define the optimal values

An optimal path to *stage* is based on an optimal path to *stage*–1

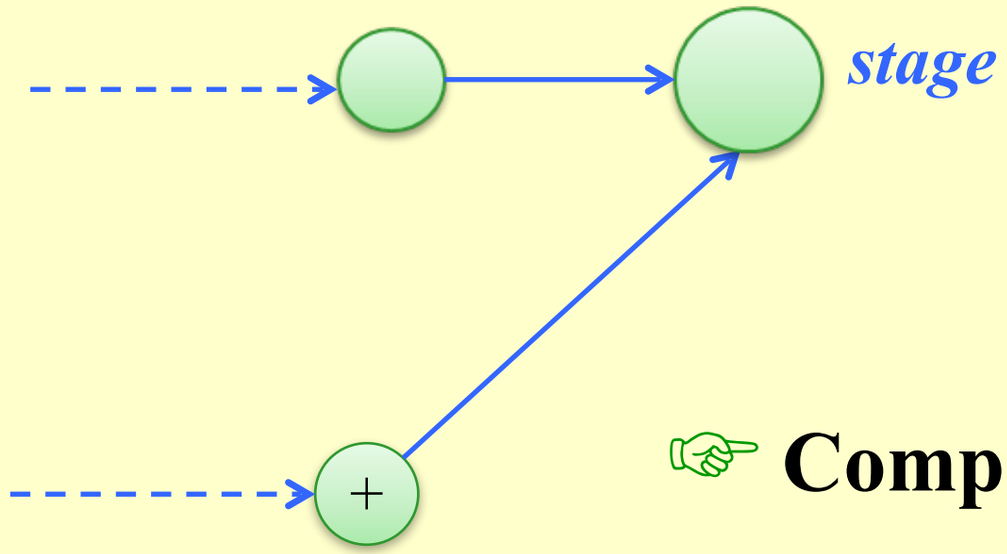


☞ Compute the values in some order

```
for (stage=1; stage<=n; stage++){
    for (line=0; line<=1; line++){
    f[line][stage] = min(
        f[  line][stage-1] + t_process[  line][stage-1],
        f[1-line][stage-1] + t_transit[1-line][stage-1]);
    }
}
```

☞ Recursively define the optimal values

An optimal path to *stage* is based on an optimal path to *stage*–1

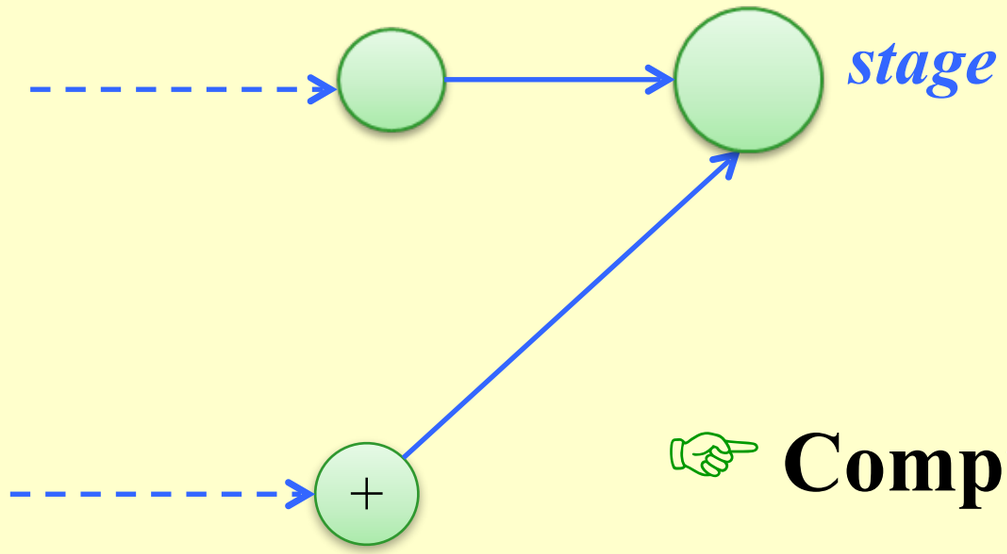


☞ Compute the values in some order

```
f[0][0]=0;  f[1][0]=0;
for (stage=1; stage<=n; stage++){
   for (line=0; line<=1; line++){
   f[line][stage] = min(
       f[  line][stage-1] + t_process[  line][stage-1],
       f[1-line][stage-1] + t_transit[1-line][stage-1]);
   }
}
```

☞ Recursively define the optimal values

An optimal path to *stage* is based on an optimal path to *stage*–1

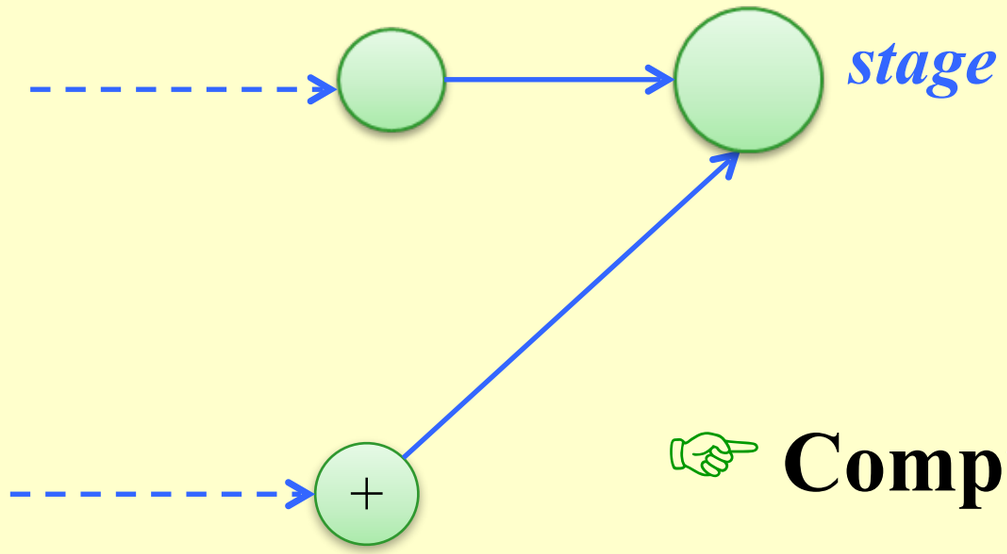


☞ Compute the values in some order

```
f[0][0]=0;  f[1][0]=0;
for (stage=1; stage<=n; stage++){
   for (line=0; line<=1; line++){
   f[line][stage] = min(
       f[  line][stage-1] + t_process[  line][stage-1],
       f[1-line][stage-1] + t_transit[1-line][stage-1]);
   }
}
Solution = min(f[0][n],f[1][n]);
```

☞ Recursively define the optimal values

An optimal path to *stage* is based on an optimal path to *stage*–1



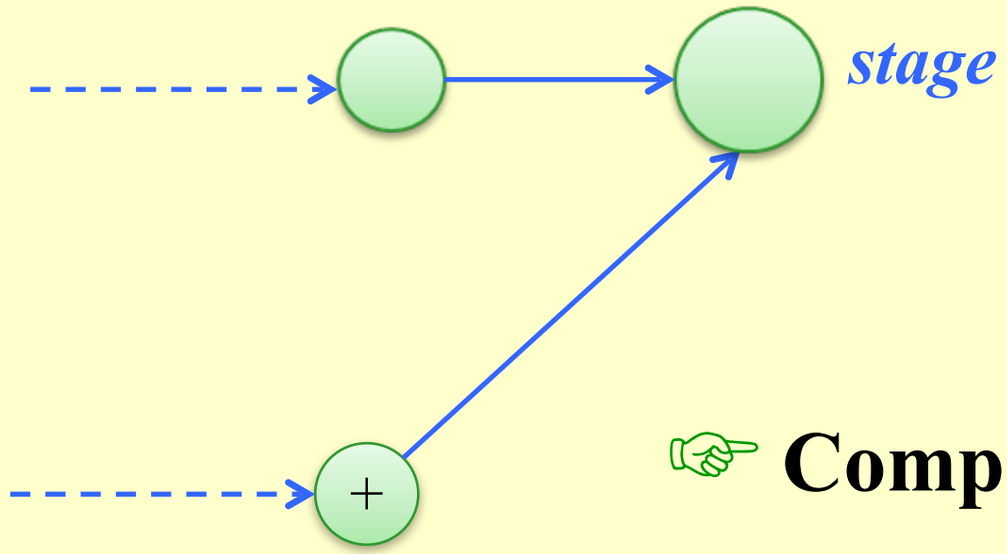
☞ Compute the values in some order

O(*N*) time + O(*N*) space

```
f[0][0]=0;  f[1][0]=0;
for (stage=1; stage<=n; stage++){
   for (line=0; line<=1; line++){
   f[line][stage] = min(
       f[  line][stage-1] + t_process[  line][stage-1],
       f[1-line][stage-1] + t_transit[1-line][stage-1]);
   }
}
Solution = min(f[0][n],f[1][n]);
```

☞ Recursively define the optimal values

An optimal path to *stage* is based on an optimal path to *stage*–1



☞ Compute the values in some order

O(*N*) time + O(*N*) space

```
f[0][0]=0;  f[1][0]=0;
for (stage=1; stage<=n; stage++){
   for (line=0; line<=1; line++){
   f[line][stage] = min(
       f[  line][stage-1] + t_process[  line][stage-1],
       f[1-line][stage-1] + t_transit[1-line][stage-1]);
   }
}
Solution = min(f[0][n],f[1][n]);
```

☞ Reconstruct the solving strategy

☞ Reconstruct the solving strategy

```
f[0][0]=0;
f[1][0]=0;
for(stage=1; stage<=n; stage++){
  for(line=0; line<=1; line++){
    f_stay = f[  line][stage-1] + t_process[  line][stage-1];
    f_move = f[1-line][stage-1] + t_transit[1-line][stage-1];
    if (f_stay<f_move){
      f[line][stage] = f_stay;

    }
    else {
      f[line][stage] = f_move;

    }
  }
}
```

☞ Reconstruct the solving strategy

```
f[0][0]=0;
f[1][0]=0;
for(stage=1; stage<=n; stage++){
  for(line=0; line<=1; line++){
    f_stay = f[  line][stage-1] + t_process[  line][stage-1];
    f_move = f[1-line][stage-1] + t_transit[1-line][stage-1];
    if (f_stay<f_move){
      f[line][stage] = f_stay;
      L[line][stage] = line;
    }
    else {
      f[line][stage] = f_move;

    }
  }
}
```

☞ Reconstruct the solving strategy

```
f[0][0]=0;
f[1][0]=0;
for(stage=1; stage<=n; stage++){
  for(line=0; line<=1; line++){
    f_stay = f[  line][stage-1] + t_process[  line][stage-1];
    f_move = f[1-line][stage-1] + t_transit[1-line][stage-1];
    if (f_stay<f_move){
      f[line][stage] = f_stay;
      L[line][stage] = line;
    }
    else {
      f[line][stage] = f_move;
      L[line][stage] = 1-line;
    }
  }
}
```

☞ Reconstruct the solving strategy

```
f[0][0]=0; L[0][0]=0;
f[1][0]=0; L[1][0]=0;
for(stage=1; stage<=n; stage++){
  for(line=0; line<=1; line++){
    f_stay = f[  line][stage-1] + t_process[  line][stage-1];
    f_move = f[1-line][stage-1] + t_transit[1-line][stage-1];
    if (f_stay<f_move){
      f[line][stage] = f_stay;
      L[line][stage] = line;
    }
    else {
      f[line][stage] = f_move;
      L[line][stage] = 1-line;
    }
  }
}
```

☞ Reconstruct the solving strategy

```
f[0][0]=0; L[0][0]=0;
f[1][0]=0; L[1][0]=0;
for(stage=1; stage<=n; stage++){
  for(line=0; line<=1; line++){
    f_stay = f[  line][stage-1] + t_process[  line][stage-1];
    f_move = f[1-line][stage-1] + t_transit[1-line][stage-1];
    if (f_stay<f_move){
      f[line][stage] = f_stay;
      L[line][stage] = line;
    }
    else {
      f[line][stage] = f_move;
      L[line][stage] = 1-line;
    }
  }
}
```

```
line = f[0][n]<f[1][n]?0:1;
for(stage=n; stage>0; stage--){
  plan[stage] = line;
  line = L[line][stage];
}
```

Elements of DP:

Elements of DP:

- ☞ **Optimal substructure**

Elements of DP:

- ☞ **Optimal substructure**
- ☞ **Overlapping sub-problems**

Elements of DP:

- Optimal substructure
- Overlapping sub-problems

Discussion 11:
When *can't* we apply dynamic programming?

Elements of DP:

- ☞ Optimal substructure
- ☞ Overlapping sub-problems

Discussion 11:
When *can't* we apply dynamic programming?

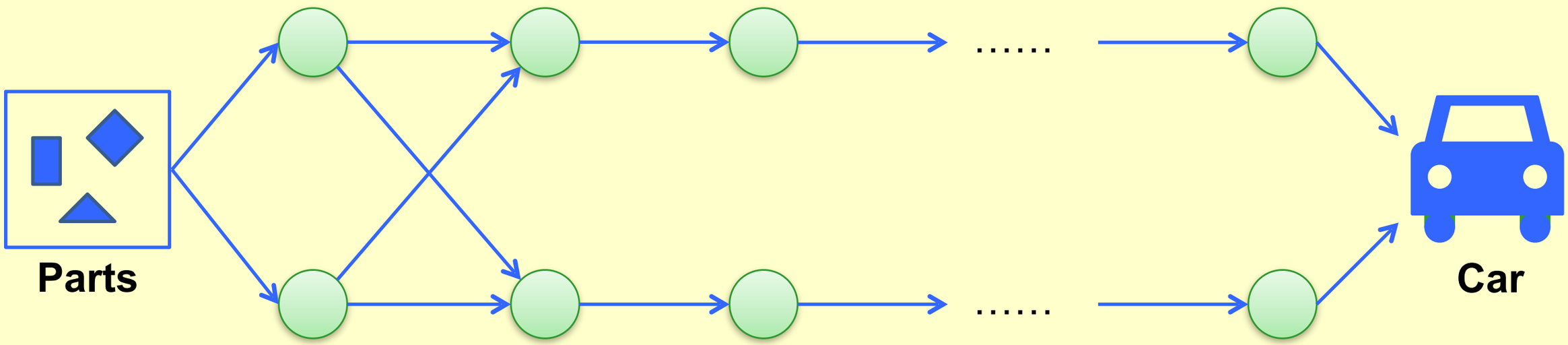
- ☞ History-dependency

Elements of DP:

- ☞ Optimal substructure
- ☞ Overlapping sub-problems

Discussion 11:
When *can't* we apply dynamic programming?

History-dependency

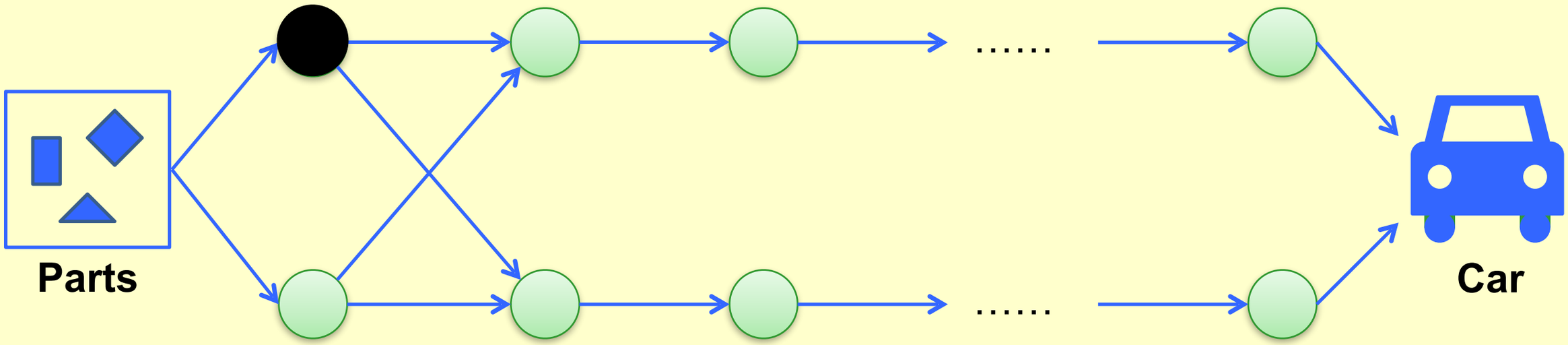


Elements of DP:

- ☞ Optimal substructure
- ☞ Overlapping sub-problems

Discussion 11:
When *can't* we apply dynamic programming?

History-dependency

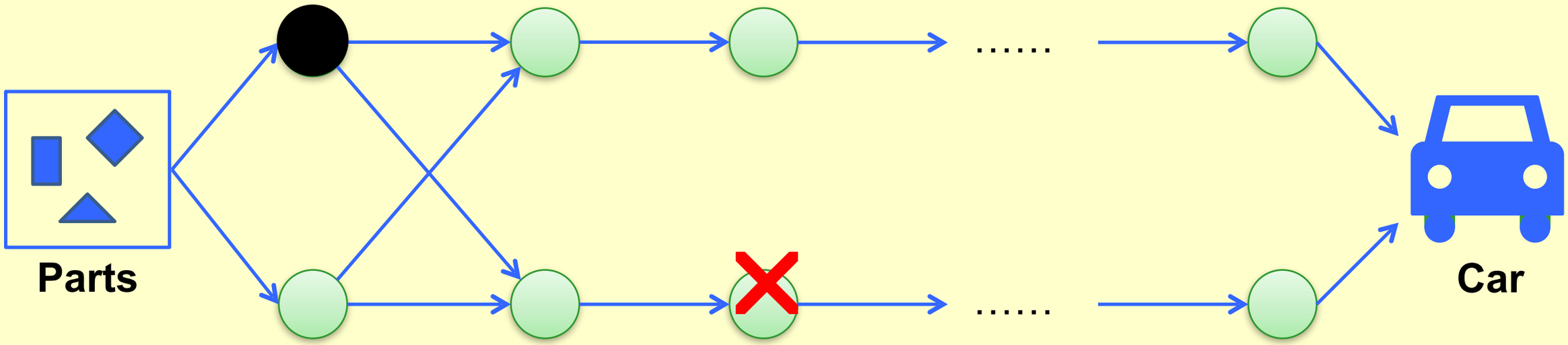


Elements of DP:

- ☞ Optimal substructure
- ☞ Overlapping sub-problems

Discussion 11:
When *can't* we apply dynamic programming?

History-dependency

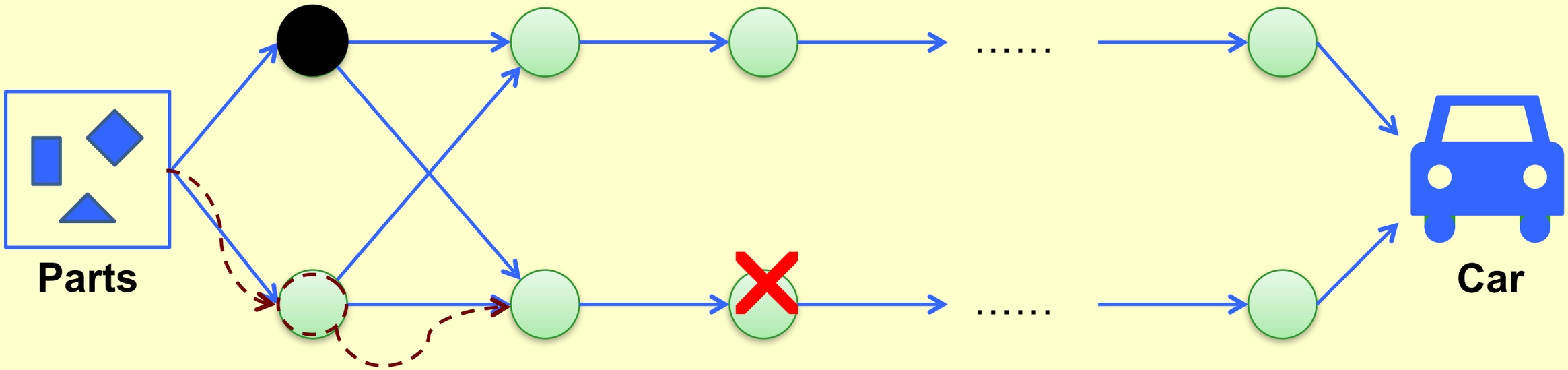


Elements of DP:

- ☞ Optimal substructure
- ☞ Overlapping sub-problems

Discussion 11:
When *can't* we apply dynamic programming?

History-dependency

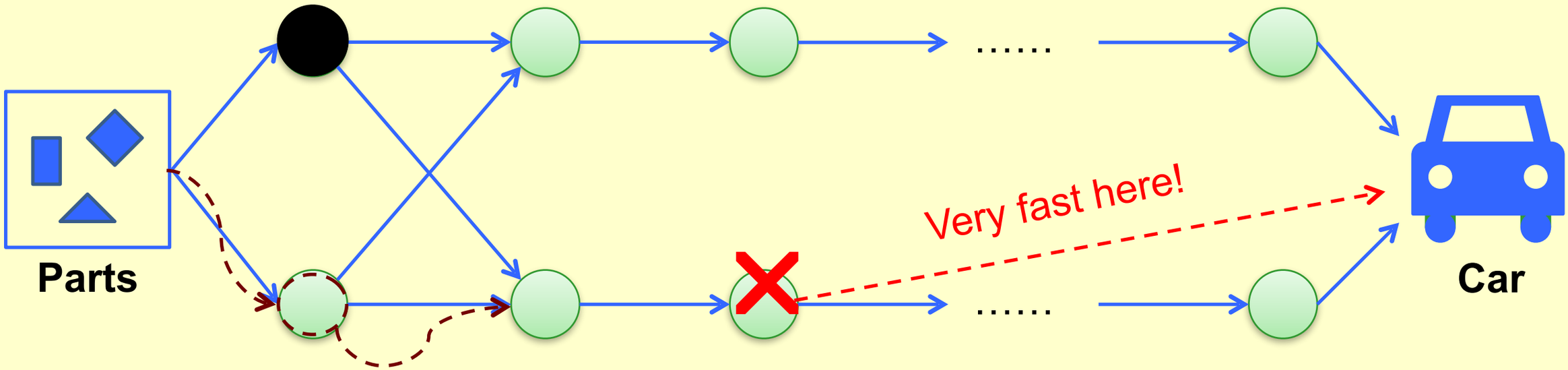


Elements of DP:

- ☞ Optimal substructure
- ☞ Overlapping sub-problems

Discussion 11:
When *can't* we apply dynamic programming?

History-dependency

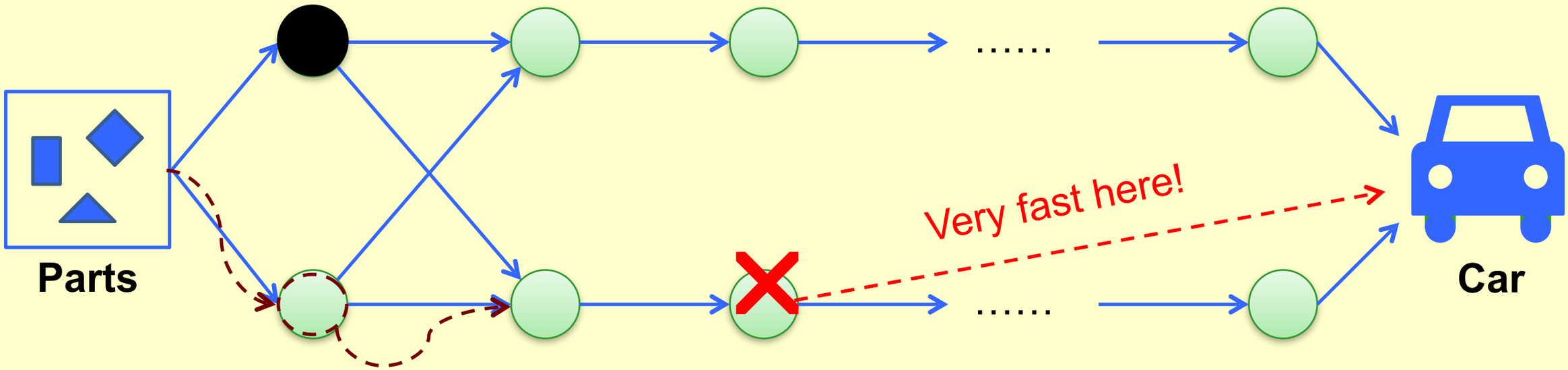


Elements of DP:

- ☞ **Optimal substructure**
- ☞ **Overlapping sub-problems**

Discussion 11:
When *can't* we apply dynamic programming?

History-dependency



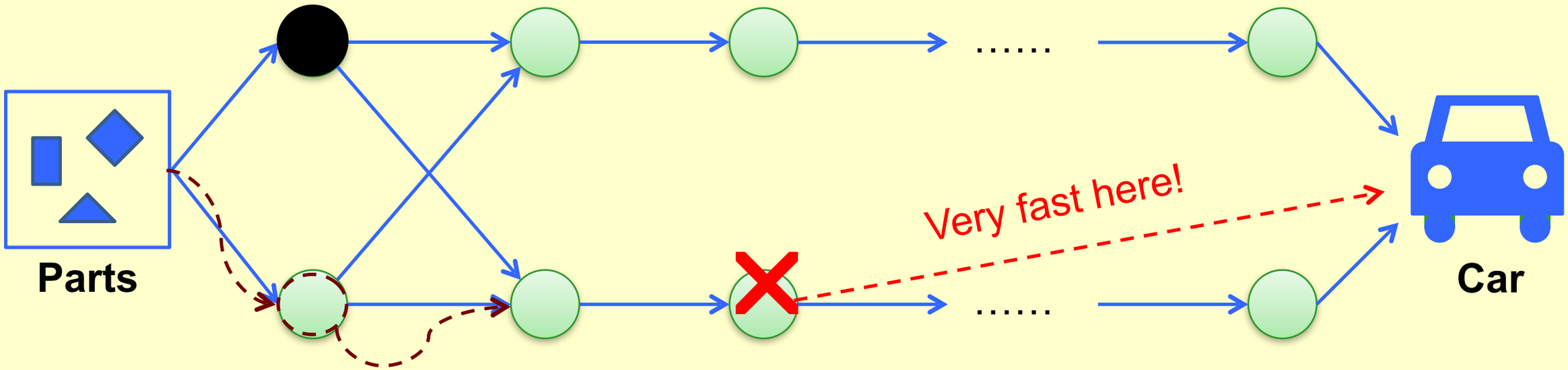
Elements of DP:

- Optimal substructure
- Overlapping sub-problems

If sub-problems do not overlap…
It becomes sort of pointless

Discussion 11:
When *can't* we apply dynamic programming?

History-dependency



Research Project 4
Red-black Tree (26)

Research Project 4
Red-black Tree (26)

How many distinct red-black trees are there that consist of exactly *N* internal nodes?

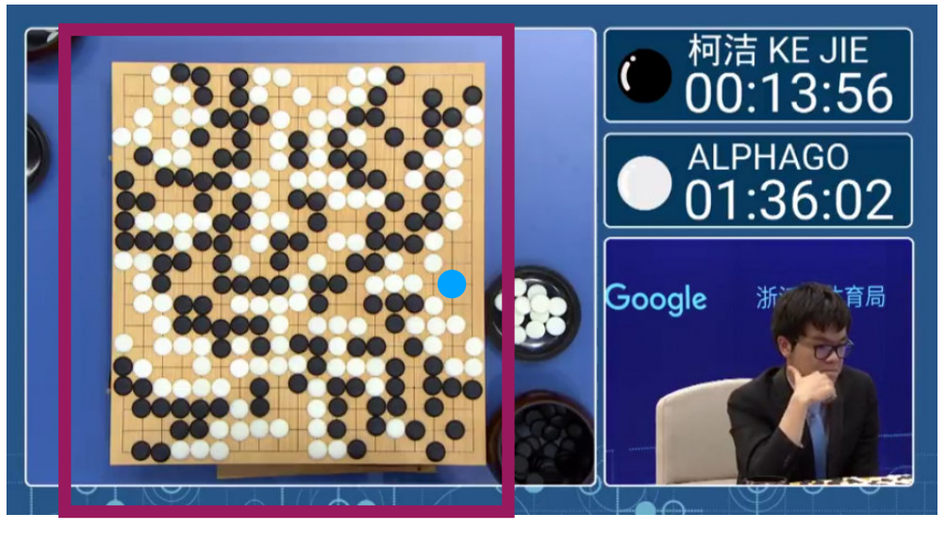
Detailed requirements can be downloaded from

https://pintia.cn/

Outline: Dynamic Programming

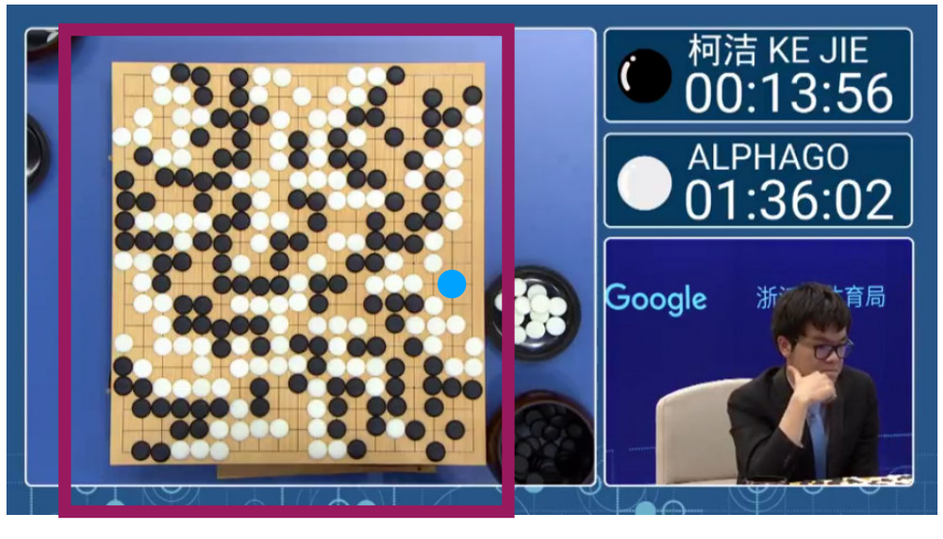
- Introduction to DP
- Solving Markov decision process
- Reinforcement learning
- Take-home messages

The Decision Problem



- The agent faces with a series of "states".
- Need to choose the corresponding "actions".
- Each action has a utility/cost.
- Target: maximize the total reward in a decision sequence by always choosing the right action.

The Decision Problem



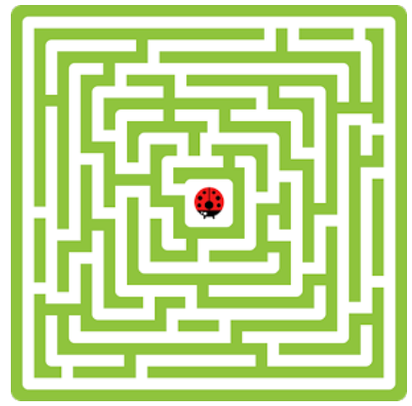
- The agent faces with a series of "states".
- Need to choose the corresponding "actions".
- Each action has a utility/cost.

In this lecture,
we given utility a name: reward.

- Target: maximize the total reward in a decision sequence by always choosing the right action.

Markov Decision Process

- How to model a maze problem?
- State: the current position.
- Action: left, right, up, down.



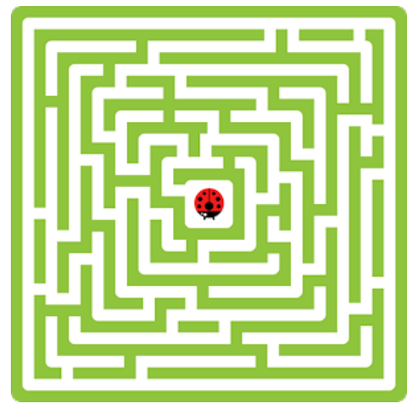
- Transition: where is the next position when take an action?
- Reward: how good is it instantly when take an action?
- Discount factor: How much the current action influences future?

Markov decision process (MDP) is the decision making model in RL with specific assumptions.

Mathematical Formulation

- A Markov Decision Process (MDP) is a five-tuple

$$< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma >$$



- $\mathcal{S}$ — The space of possible states (cont. or discrete)
- $\mathcal{A}$ — The space of possible actions (cont. or discrete)
- $\mathcal{P} : p(s_{t+1}|s_t, a_t)$ — The transition function (distribution)
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$ — The reward function
- γ — The discount factor of rewards

The transition and reward functions can be stochastic!

The Markov Property

"The future is independent of the past given the present"

- The Markov property:

$$p(s_{t+1}|s_t, a_t) = p(s_{t+1}|s_1, s_2, ...s_t, a_t)$$

- The next state is only decided by the current state and action.
- The current state is a sufficient statistic.
- Non-Markovian decision problem:

小张出生于中国，2016年来到美国留学。小张的母语是(？)

The Learning Agent

- The agent takes a series of actions, experiences a series of states, and receives a series of rewards:

$$\{s_1, a_1, r_1\}, \{s_2, a_2, r_2\}, \{s_3, a_3, r_3\}...$$

- policy: function $p(a|s)$ used to select actions on any states.

- The target is to find the optimal policy to maximize the discounted total reward along the timeline:

$$r_1 + \gamma r_2 + \gamma^2 r_3 + ... = \sum_{t=1}^{\infty} \gamma^{t-1} r_t$$

- The discount factor measures how much the long term effect of the current action is concerned.

Value Functions: State Value Function V

- The state value function of a given policy is the expected total reward start from a given state, then follow the policy:

$$V_\pi(s) = \mathbb{E}_{\pi, p(s|s,a)}\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s\right]$$

- The optimal policy have the optimal value function:

$$V_{\pi^*}(s) = \max_\pi V_\pi(s)$$

Value Functions: Action-State Value Function Q

- The action-state value function of a given policy is the expected total reward start from a given state, execute a given action, then follow the policy:

$$Q_\pi(s,a) = \mathbb{E}_{\pi, p(s|s,a)}\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a\right]$$

- The optimal deterministic policy chooses the optimal action:

$$\pi^*(s) = \arg\max_a Q_{\pi^*}(s,a)$$

If the optimal action-state value function is known, so is the optimal policy!

Bellman Equation

- For the state value function,

$$V_\pi(s) = \mathbb{E}_{\pi(s),p(s|s,a)}\Big[\sum_{t=0}^{\infty}\gamma^t r_t|s_0 = s\Big]$$

$$= \mathbb{E}_{\pi(s_0),p(s_1|s_0,a_0)}\Big[r_0 + \gamma\mathbb{E}_{\pi(s),p(s|s,a)}\big[\sum_{t=1}^{\infty}\gamma^{t-1} r_t|s_1\big]|s_0 = s\Big]$$

$$= \mathbb{E}_{\pi(s_0),p(s_1|s_0,a_0)}\Big[r_0 + \gamma V_\pi(s_1)|s_0 = s\Big]$$

Recursive Definition

- For discrete state and action, and deterministic policy,

$$V_\pi(s) = \sum_{s'} p\big(s'|s,\pi(s)\big)\Big[r\big(s,\pi(s),s'\big) + \gamma V_\pi(s')\Big]$$

Bellman Equation (Cont.)

- For the action-state value function,

$$Q_\pi(s,a) = \mathbb{E}_{\pi(s),p(s|s,a)}\Big[\sum_{t=0}^{\infty}\gamma^t r_t|s_0=s,a_0=a\Big]$$

$$= \mathbb{E}_{\pi(s_0),p(s_1|s_0,a_0)}\Big[r_0+\gamma\mathbb{E}_{\pi(s),p(s|s,a)}\big[\sum_{t=1}^{\infty}\gamma^{t-1}r_t|s_1,a_1\big]|s_0=s,a_0=a\Big]$$

$$= \mathbb{E}_{\pi(s_0),p(s_1|s_0,a_0)}\Big[r_0+\gamma Q_\pi(s_1,a_1)|s_0=s,a_0=a\Big]$$

Recursive Definition

- For discrete state and action, and deterministic policy,

$$Q_\pi(s,a) = \sum_{s'} p\big(s'|s,a\big)\Big[r(s,a,s')+\gamma Q_\pi\big(s',\pi(s')\big)\Big]$$

Bellman Equation (Cont.)

- For optimal deterministic policy π^*,

$$V_{\pi^*}(s) = \max_a \Big[\sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma V_{\pi^*}(s')] \Big]$$

$$Q_{\pi^*}(s,a) = \sum_{s'} p(s'|s,a) \Big[r(s,a,s') + \gamma \max_{a'} Q_{\pi^*}(s',a') \Big]$$

- Then the optimal deterministic policy is

$$\pi^*(s) = \arg\max_a Q_{\pi^*}(s,a)$$

- Due the recursive structure, the optimal value functions can be solved by dynamical programming. This assumes that the full information of the MDP is known!

Solving MDP: Value Iteration

- Initialize value function V_0
- For i=1,2,3… until convergence
    - Update V_i for each state

$$V_i(s) = \max_a \left[\sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma V_{i-1}(s')] \right]$$

- Theoretical convergence guarantee to V^* and π^*

Solving MDP:Value Iteration

- Initialize value function V_0
- For i=1,2,3… until convergence
    - Update V_i for each state

Why iterative update?
Loop exists in the MDP!

$$V_i(s) = \max_a \left[\sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma V_{i-1}(s')] \right]$$

- Theoretical convergence guarantee to V^* and π^*

Solving MDP: Policy Iteration

- Initialize value function V_0 and policy π_0
- For i=1,2,3… until convergence
    - Policy evaluation step: update V_i for each state

$$V_i(s) = r + \gamma V_{i-1}^{\pi_{i-1}}(s')$$

    - Policy improvement step: update π_i for each s-a pair.

$$\pi(s) \leftarrow arg\max_a \sum_{s',r'} p(s', r|s, a)[r + \gamma V(s')]$$

- Theoretical convergence guarantee to V^* and π^*

Solving MDP: Policy Iteration

- Initialize value function V_0 and policy π_0
- For i=1,2,3… until convergence
  - Policy evaluation step: update V_i for each state

$$V_i(s) = r + \gamma V_{i-1}^{\pi_{i-1}}(s')$$

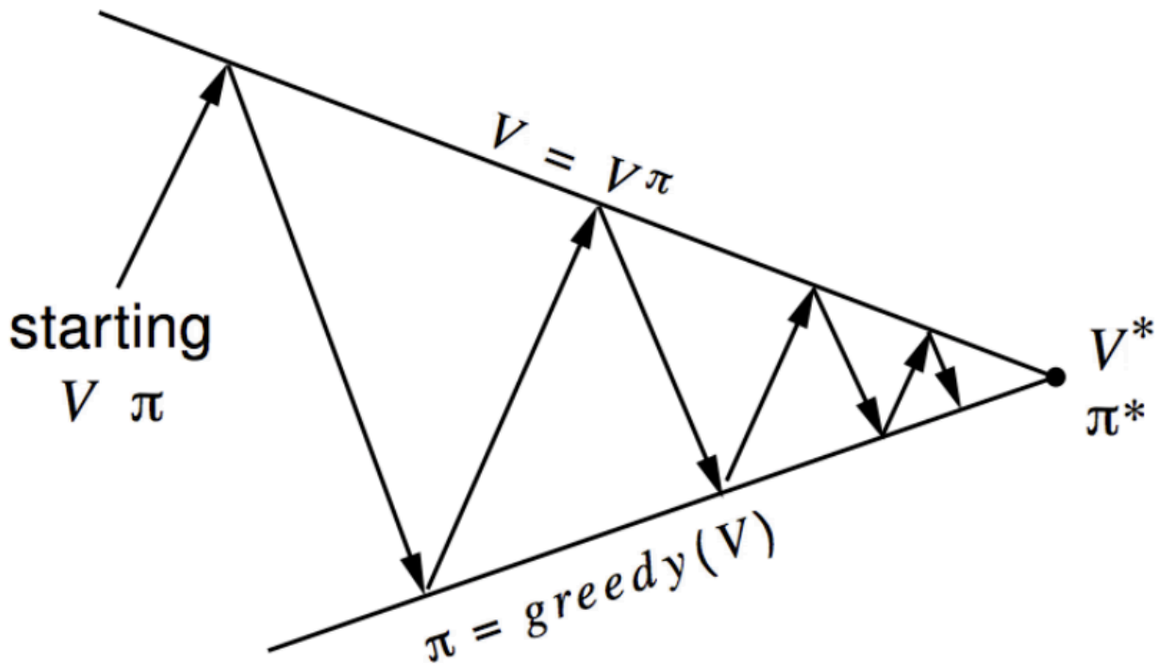
Calculated based on π_{i-1}
Actually an inner loop to do iterative update until convergence

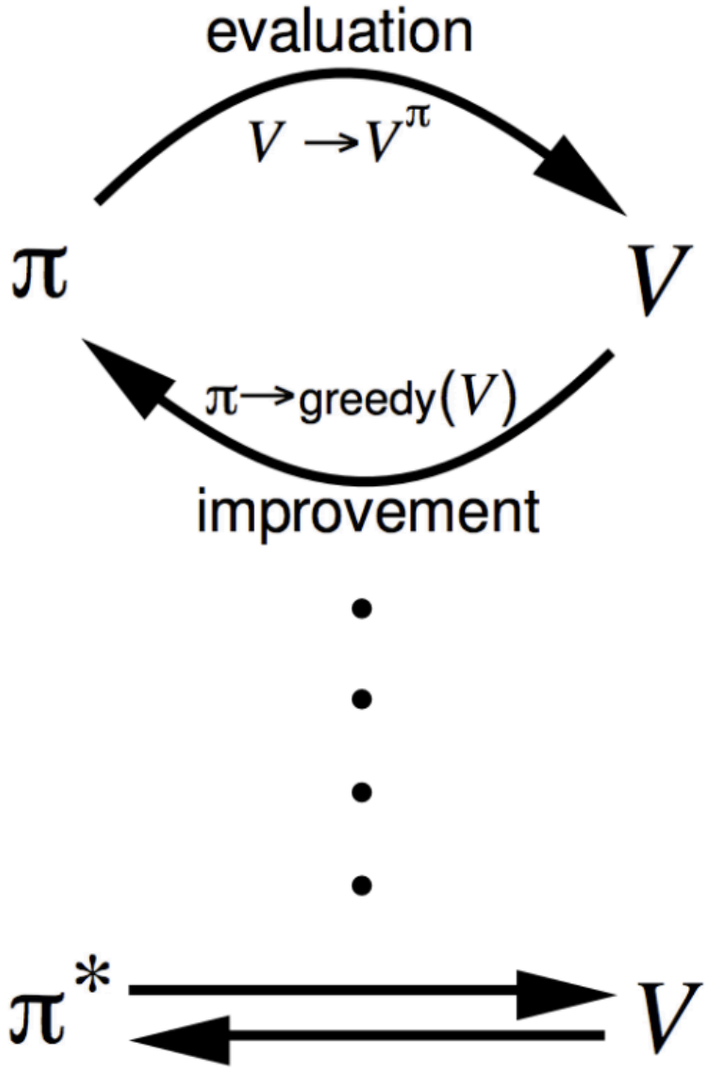
  - Policy improvement step: update π_i for each s-a pair.

$$\pi(s) \leftarrow arg\max_a \sum_{s',r'} p(s', r|s, a)[r + \gamma V(s')]$$

- Theoretical convergence guarantee to V^* and π^*

Policy Iteration (Cont.)





Policy evaluation Estimate v_π
Iterative policy evaluation

Policy improvement Generate $\pi' \geq \pi$
Greedy policy improvement

Slide courtesy: David Silver

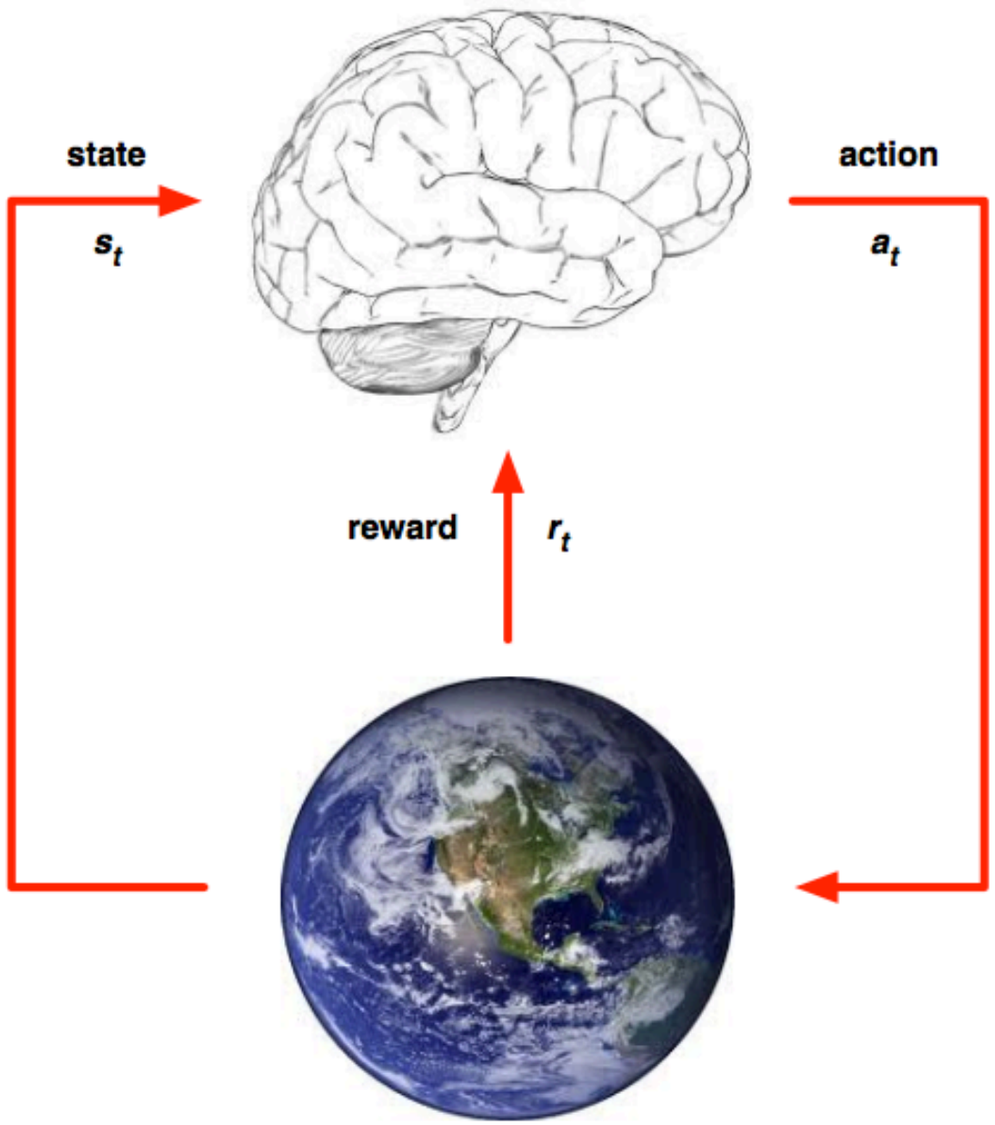
Outline: Dynamic Programming

- Introduction to DP
- Solving Markov decision process
- Reinforcement learning
- Take-home messages

Reinforcement Learning

- When full information of the MDP is known, the value function can be solved by planning.
- But how to solve when not fully known? $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma >$
- In RL, usually the state transition $\mathcal{P}$ and reward function $\mathcal{R}$ are not known.
- The agent has to learn by trial and error, facing with the exploration and exploitation problem.

Agent and Environment

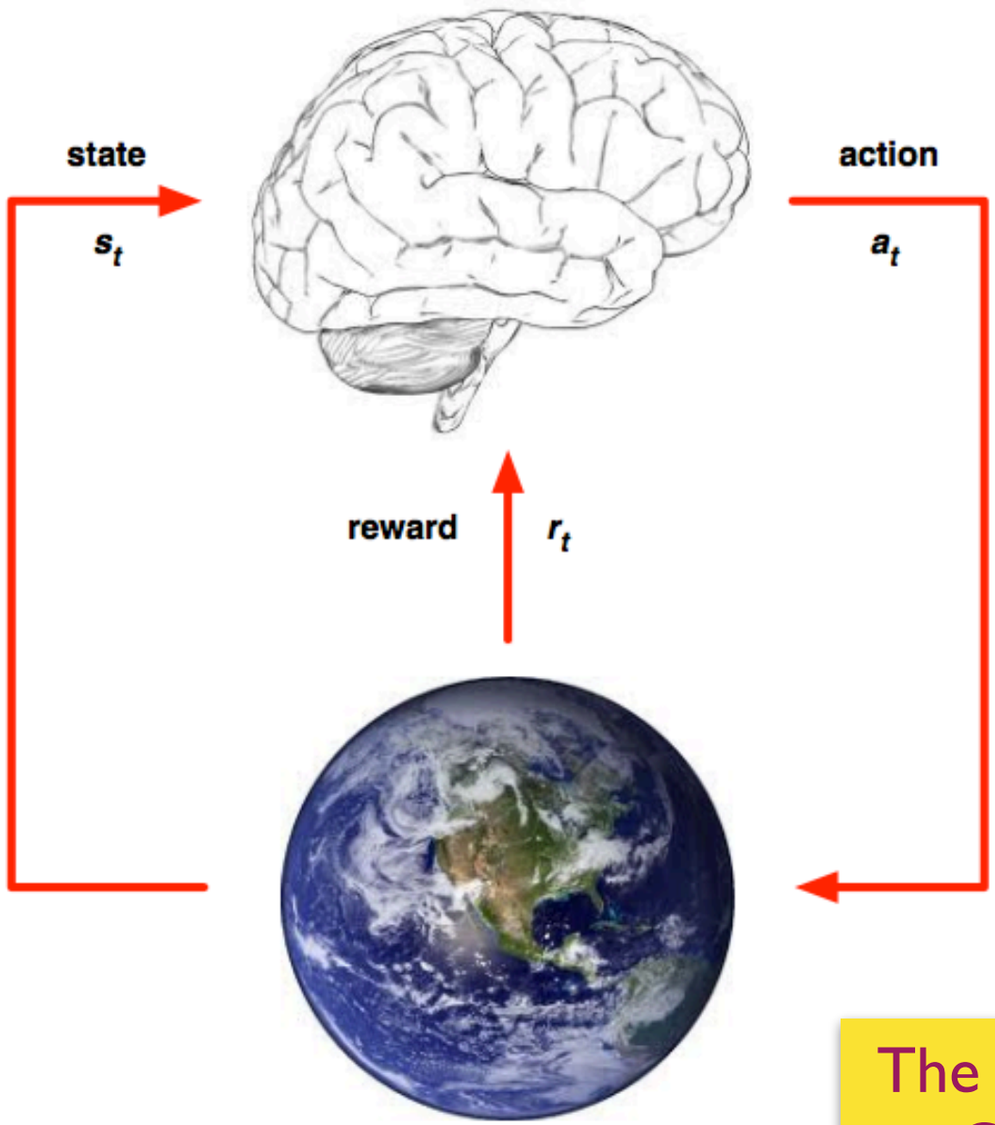


- At each step t the agent:
  - Receives state s_t
  - Receives scalar reward r_t
  - Executes action a_t
- The environment:
  - Receives action a_t
  - Emits state s_t
  - Emits scalar reward r_t

- The target is still to learn the optimal value function.

Slide courtesy: David Silver

Agent and Environment



- At each step t the agent:
  - Receives state s_t
  - Receives scalar reward r_t
  - Executes action a_t
- The environment:
  - Receives action a_t
  - Emits state s_t
  - Emits scalar reward r_t

The agent can only interact with true environment.
Can not use model for searching or planning.

- The target is still to learn the optimal value function.

Slide courtesy: David Silver

Basic Idea

- Value function based RL aims at estimating the optimal value function.

- Value function update: update using new estimation

$$Q_{i+1}(s,a) = (1-\alpha)Q_i(s,a) + \alpha\hat{Q}_i(s,a)$$

  - Monte-Carlo RL — Estimate by sampled trajectories

  - Temporal difference Learning — SARSA and Q-learning.

- Policy Improvement:

  - Based on new value function, with ϵ - greedy.

Monte-Carlo RL

- Given policy π_i, we can sample trajectories:

$$\{s_1, a_1, r_1\}, \{s_2, a_2, r_2\}, \{s_3, a_3, r_3\}...$$

- Then we can get empirical estimate:

$$\hat{Q}_i(s_1, a_1) = r_1 + \gamma r_2 + \gamma^2 r_3 + ...$$

Can we still update the policy in greedy?

No!

- Update value function:

$$Q_{i+1}(s, a) = (1 - \alpha)Q_i(s, a) + \alpha\hat{Q}_i(s, a)$$

- Follow the spirit of policy iteration, update $\pi_i \rightarrow \pi_{i+1}$

Exploration vs. Exploitation

"Behind one door is tenure - behind the other is flipping burgers at McDonald's."

- There are two doors in front of you.
- You open the left door and get reward 0
  $V(left) = 0$
- You open the right door and get reward $+1$
  $V(right) = +1$
- You open the right door and get reward $+3$
  $V(right) = +2$
- You open the right door and get reward $+2$
  $V(right) = +2$

$\vdots$

- Are you sure you've chosen the best door?

Slide courtesy by David Silver

Exploration vs. Exploitation (Cont.)

- In policy iteration, the policy improvement step is greedy:

$$\pi_i(s) = \arg\max_a Q_i(s, a)$$

- But for RL, since the environment is not fully known, greedy update may perform arbitrarily bad — need to allow some exploration.

- Common choice: use ϵ- greedy policy:

with prob. $1 - \epsilon$, execute as greedy

with prob. ϵ, execute randomly

- Theoretical guarantee: If the exploration vanishes, we can ensure convergence.

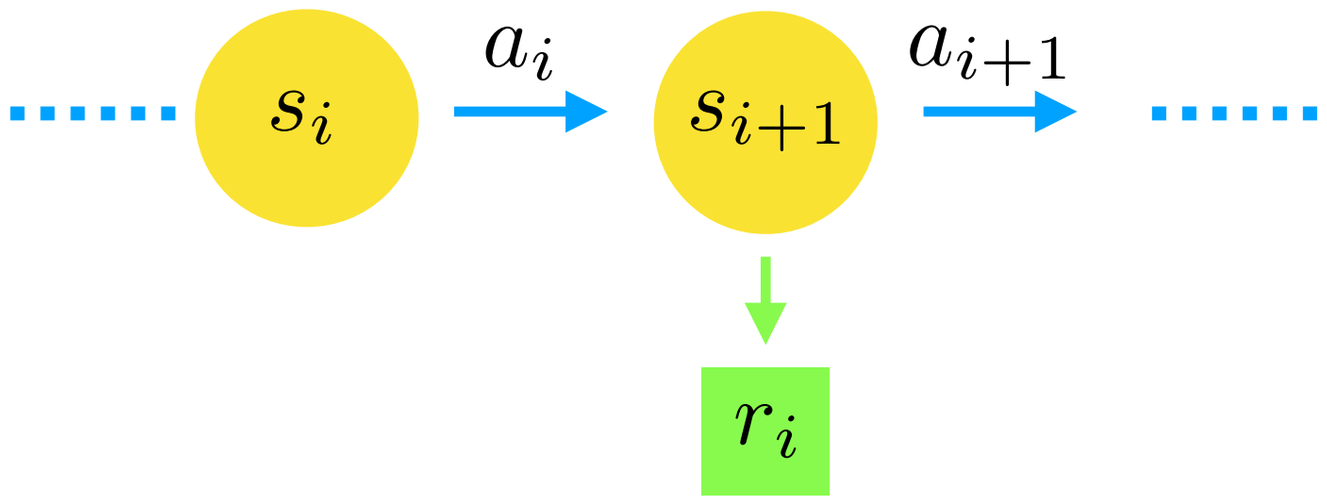
TD vs. MC

- Temporal-difference (TD) learning has several advantages over Monte-Carlo (MC)
  - Lower variance
  - Online
  - Incomplete sequences
- Natural idea: use TD instead of MC in our control loop
  - Apply TD to $Q(S, A)$
  - Use ϵ-greedy policy improvement
  - Update every time-step

Slide courtesy: David Silver

SARSA

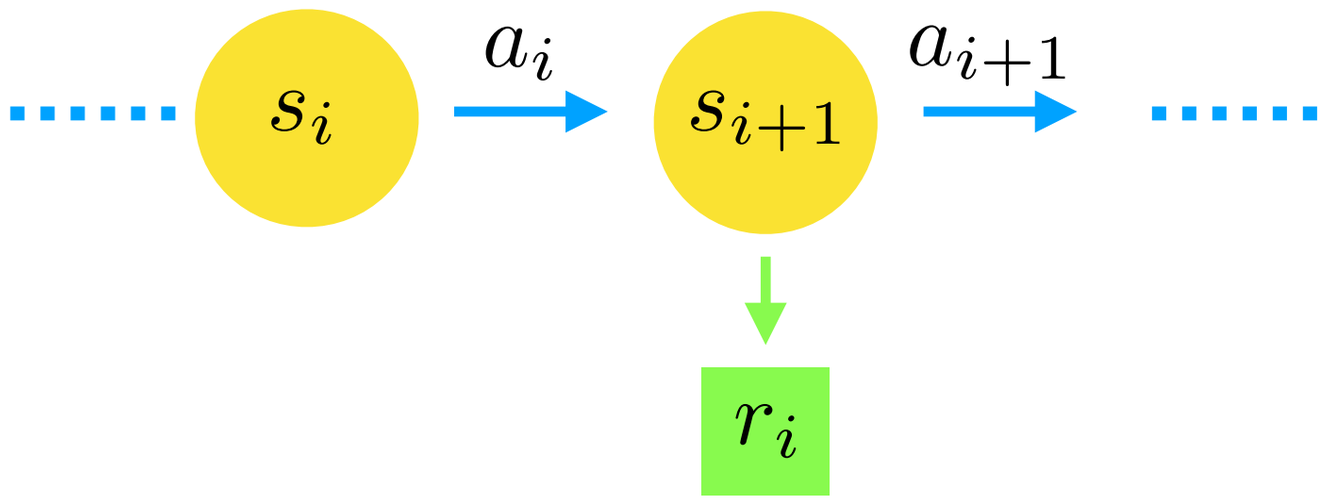
- "State-Action-Reward-State-Action" — SARSA



Run the current best ϵ-greedy policy

SARSA

- "State-Action-Reward-State-Action" — SARSA



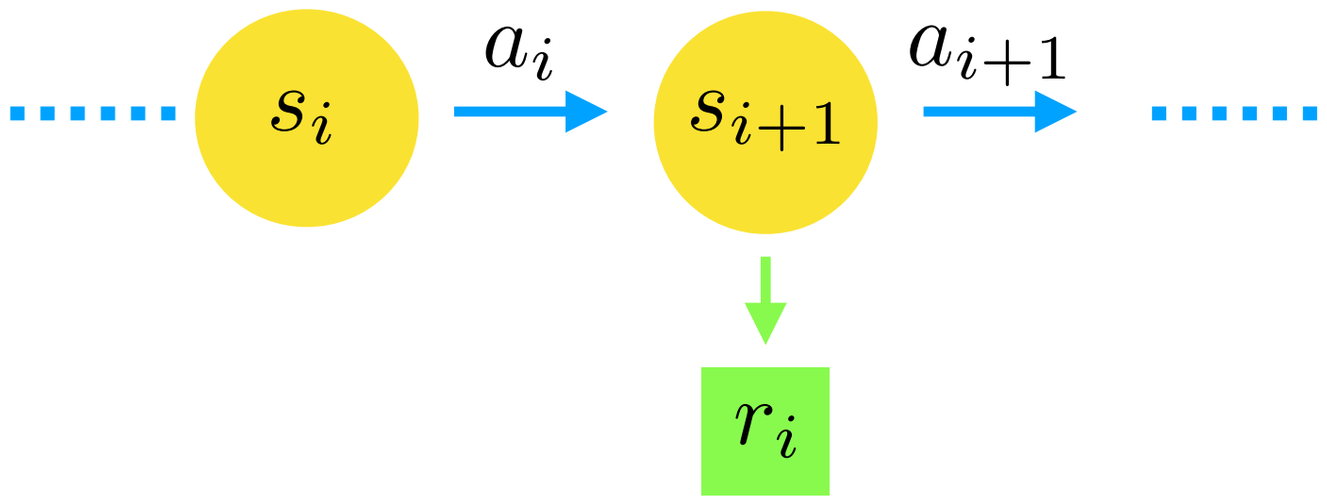
Run the current best ϵ-greedy policy

- Once collect s-a-r-s-a sample, do value function update:

$$Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \alpha \Big[r_i + \gamma Q(s_{i+1}, a_{i+1}) - Q(s_i, a_i) \Big]$$

SARSA

- "State-Action-Reward-State-Action" — SARSA



Run the current best ϵ-greedy policy

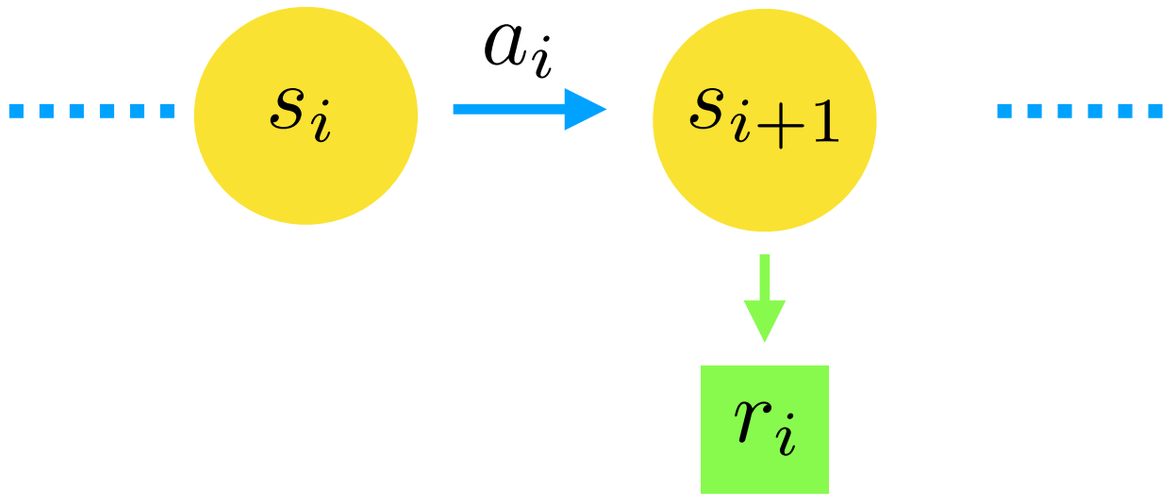
- Once collect s-a-r-s-a sample, do value function update:

$$Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \alpha \Big[r_i + \gamma Q(s_{i+1}, a_{i+1}) - Q(s_i, a_i) \Big]$$

TD error

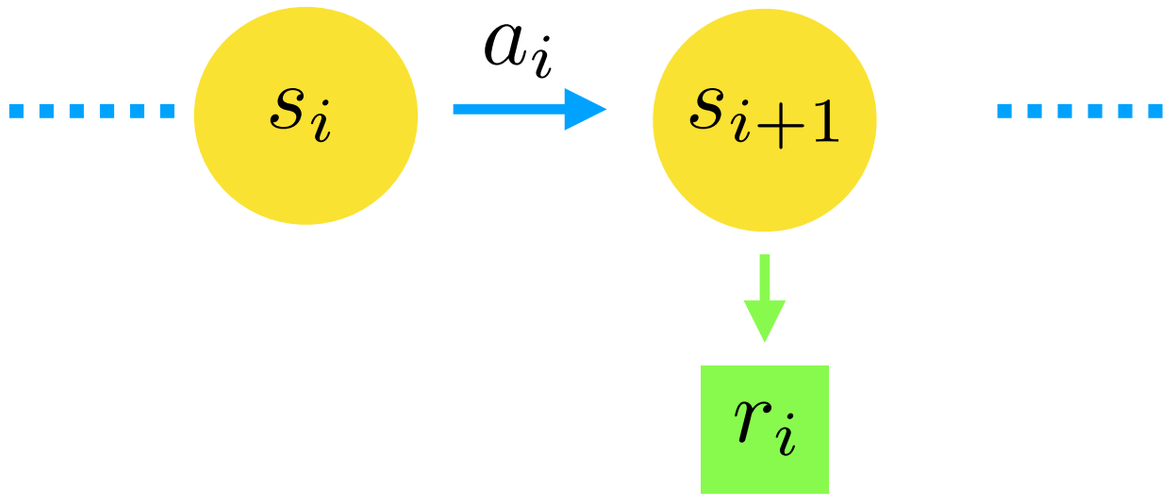
Always use the policy on-the-run, called "on-policy"

Q-Learning



Run the current
best ϵ-greedy policy

Q-Learning



Run the current
best ϵ-greedy policy

- Once collect s-a-r-s sample, do value function update:

$$Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \alpha\Big[r_i + \gamma Q(s_{i+1}, \hat{\pi}^*(s_{i+1})) - Q(s_i, a_i)\Big]$$

The current best policy

The policy on the run can be different from
the current best policy in the update, called "off-policy"

Outline: Dynamic Programming

- Introduction to DP
- Solving Markov decision process
- Reinforcement learning
- Take-home messages

Those who cannot remember the past are condemned to repeat it.

- Dynamic Programming

Those who cannot remember the past are condemned to repeat it.

- Dynamic Programming

Try to do more exercises.

Thanks for your attention!
Discussions?

Reference

Data Structure and Algorithm Analysis in C (2nd Edition)： Sec. 10.3.

Introduction to Algorithms (4th Edition): Chap.14.

https://www.davidsilver.uk/teaching/ Lec. 1-5